ENHANCING HOST BASED INTRUSION DETECTION SYSTEMS WITH DANGER THEORY OF ARTIFICIAL IMMUNE SYSTEMS

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

Suhair Hafez Amer

Certificate of Approval:

Saad Biaz
Associate Professor
Computer Science and Software Engineering

Drew Hamilton, Chair
Associate Professor
Computer Science and Software Engineering

Richard Chapman
Associate Professor
Computer Science and Software Engineering

Levent Yilmaz
Assistant Professor
Computer Science and Software Engineering

Joe F. Pittman
Interim Dean
Graduate School

ENHANCING HOST BASED INTRUSION DETECTION SYSTEMS WITH DANGER THEORY OF ARTIFICIAL IMMUNE SYSTEMS

Suhair Hafez Amer

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama
May 10, 2008

ENHANCING HOST BASED INTRUSION DETECTION SYSTEMS WITH DANGER

THEORY OF ARTIFICIAL IMMUNE SYSTEMS

Suhair Hafez Amer

Signature of Author

Date of Graduation

DISSERTATION ABSTRACT

ENHANCING HOST BASED INTRUSION DETECTION SYSTEMS WITH DANGER THEORY OF ARTIFICIAL IMMUNE SYSTEMS

Suhair Hafez Amer

Doctor of Philosophy, May 10, 2008
(M.S., American University in Cairo, 2000)
(B.S., American University in Cairo, 1998)

302 Typed Pages

Directed by Drew Hamilton, Jr.

Rather than discriminating activity by belonging to self or non-self, danger theory extends its discrimination to be between non-self but harmless and self but harmful. The danger theory states that the system does not respond only to foreignness (non-self) but to danger signals. In this dissertation, three methods performing host-based anomaly intrusion detection that use trails of system calls have been implemented and investigated. One system (the lookahead-pairs method based IDS) was then enhanced by incorporating danger theory mechanisms to its original design. The research consisted of two stages. In the first stage, three intrusion detection systems (IDSs) have been implemented based on the following methods: the sequence profile method, the lookahead-pairs methods, and overlap-relationship method. All systems were unable to detect the system-call-denial-of-service attack and the lookahead-pairs method had the smallest storage requirements.

In the second stage, the lookahead-pairs method based IDS has been enhanced with functionalities of the danger theory. The original lookahead-pairs method based IDS can only detect intrusions resulting from mismatch instances. In addition to detecting mismatches, the enhanced system considered the danger signals resulting from high usages of CPU and memory while in detection mode. Parameters corresponding to danger signals can be easily modified or added to our system. The lookahead pairs method enhanced with danger theory IDS had better detection rate, false positive rate and false negative rate. Both systems finished their detection stage in less than one second. Furthermore, when the lookahead pairs method based IDS is only enhanced with the iDC functionality, it will not experience any significant additional storage costs. However, if the B cell functionality is added, the storage cost would double. The systems were tested against the databases obtained from the university of New Mexico and in specific the datasets of the both the “login” and “ps” applications. In addition, different test cases were created to test the functionalities of the modified system. The implemented systems were also validated and verified and passed these tests.

ACKNOWLEDGMENTS

I thank God for giving me the well and the strength to accomplish my PhD.

I sincerely thank my advisor Dr. Drew Hamilton for his understanding and for providing me with the opportunity to pursue my Ph.D. dream. I hold great admiration for his continuous efforts to support his students and his dedication to his work.

Guidance from my chair and committee members Dr. Biaz, Dr. Chapman, and Dr. Yilmaz were very valuable and worth pursuing. I thank and I will miss the members of The Information Assurance Center (IAC), faculty, staff and students of the department of Computer Science and Software Engineering (CSSE). The engineers at both CSSE and Engineering Network Services were extremely prompt through my innumerable queries. This work was supported, in part, by: Grants NSF Due 0516432, and is gratefully acknowledged.

I have been blessed with the most wonderful and supportive family any one can wish for. There is little I can say to thank you all. This dissertation would not have been possible without the support, love, devotion and prayers of my parents "Hafez and Khawla". The every day kindness, support, love and encouragement by my husband "Bashar" has an enormous effect on accomplishing our dream. I thank God for granting me two miracles, my beloved children, who provide sunshine and joy to my life. To them "Kareem and Sarah" I dedicate this dissertation. Finally, I would like to thank the members of my extended family and friends for their prayers.

Style manual or journal used: IEEE Standard

Computer software used: Microsoft Word 2007, Microsoft Excel 2007, and Microsoft Visual Studio 2005.

TABLE OF CONTENTS

LISTS OF FIGURES

LISTS OF TABLES

CHAPTER 1

INTRODUCTION

The human immune system has been successful in defending different human organs against a wide range of harmful attacks. The Danger Theory is built on the idea that the immune system not only responds to foreignness (non-self) but also to danger signals resulting from damage to cells indicated by distress signals that are sent out when cells die an unnatural death as opposed to programmed cell death.

In this dissertation, I investigated three methods of performing host-based anomaly intrusion detection to recognize malicious code execution and enhanced the performance of lookahead-pairs method based intrusion detection system (IDS) by incorporating danger theory concepts. In particular, my research involves two major stages. In the first stage, I have implemented and studied the performance of three techniques to perform host based intrusion detection using trails of system calls. The first system, the sequence profile method, creates a database with fixed length sequences of system calls of a system's normal behavior. While in detection mode the system's current behavior is checked against this database and an intrusion is flagged if a deviation is discovered. In general, running any single application will produce thousands of system calls. The second system, the lookahead-pairs methods, improves the storage requirements of the sequence method but still creates a database of pairs of <current system call, previous system calls> within a fixed length threshold. Both systems create patterns (system call sequences) of fixed length. The third system, overlap-relationship method, involves creating a database of variable length detector sets which enable better detector coverage. All methods were unable to detect the system-call-denial of service attack and the lookahead-pairs method had the smallest storage requirements.

In the second stage, I investigated how to incorporate adaptive danger theory concepts to lookahead-pairs method and investigated the enhanced system's performance. The following mechanisms have been implemented and their performance was analyzed. First, B cells are responsible for identifying bacteria signatures (deviations or intrusion signatures). Dendritic cells are responsible for sensing safe and dangerous signals and along with the identification of bacteria (intrusions) decide if the system is under attack or not. Gathered information is then sent to T cells that carry out the remaining actions of the immune system. The original lookahead-pairs method can only detect intrusions resulting from mismatch instances. The system may experience false positive instances especially when the system is not fully trained on all normal behavior. False positives result from identifying a normal behavior as an intrusion. Since the lookahead-pairs IDS relies only on mismatches, any new sequence will be flagged as an intrusion. Lookahead-pairs method enhanced with danger theory IDS improve the detection rate since it will identify more intrusions especially those that deviate generating mismatches or exceed the mismatch threshold. It will also reduce the rate of false positive and false negative. This is because an intrusion not only depends on mismatch instances but also on other factors (signals) that describe dangerous conditions. The lookahead-pairs method enhanced with iDC and DC cells do not require additional storage cost and will give better detection results. The IDS system enhanced with B cells will require double storage requirements and will become more robust. If we choose to use both negative and positive detector sets for the B cell and iDC databases, the system could be distributed to other machines.

1.1. Dissertation Hypotheses

In this dissertation we investigated and proved the following hypotheses:

- Enhanced lookahead-pairs method with iDC signal processing has better detection rate and a lower or similar false positive and false negative rates with similar space and delay costs than the original lookahead-pairs method.
- Enhanced lookahead-pairs method with Danger Theory has a better detection rate, and a lower false positives rate and false negative rate than the original lookahead-pairs method with an additional space cost and similar delay.

1.2. Dissertation Objectives and Accomplished Stages

Host based intrusion detection systems are an important tool for detecting malicious activities on a single machine. Relying on network based intrusion detection systems is not enough because many intrusions can be missed such as installing backdoor programs, Trojan horses, etc. Identifying host based intrusions can be performed by analyzing and monitoring system calls generated by application processes. In this dissertation we investigated three intrusion detection systems to better understand how host based intrusion detection is performed. The systems are: sequence method based IDS, lookahead pairs method based IDS and variable length with overlap relationship method based IDS. Then we enhanced the lookahead pairs method IDS by incorporating different functionalities of danger theory.

The objectives accomplished during the dissertation are the following:

1. Investigated human immune system theories such as negative selection and danger theory and understood their underlining mechanisms and participating cells.
2. Investigated and understood artificial immune system based intrusion detection system and in specific those based on danger theory concepts.
3. Investigated host based intrusion detection systems in general, and immunity based host based intrusion detection systems in specific.
4. Developed a framework and models explaining danger theory concepts and functionalities.
5. Implemented the following to run on windows platform:
   - Sequence method based IDS using fixed length detector set.
   - Lookahead-pairs method based IDS using fixed length detector set.
   - Variable-length-patterns-with-an-overlap-relationship method based IDS.
   - Lookahead-pairs IDS enhanced with danger-theory concepts.

1.3. Dissertation Contributions

The main contribution of this dissertation is that I developed a danger theory based IDS and proved that it outperforms the original system that does not incorporate danger theory concepts. We have implemented the lookahead-pairs method based IDS and enhanced its performance by using functionalities of danger theory. We were able to prove that the modified IDS has better detection rate, lower false positives and false negatives and little impact on performance and storage requirements

The following will explain additional contributions. First, I have re-implemented Sequence Method, Lookahead Method and Variable Length with Overlap Relationship

Method to Run on a Windows Platform. The basic steps of each method were adapted from its respective paper. The original implementation of each system was developed on Linux or UNIX machines. Some differences exist between a windows platform and UNIX platforms such as differences in language commands and data structures. My system was implemented with Microsoft Visual Studio 2005 as a win 32 console application.

Second I identified Some Limitations of Sequence, Lookahead Pairs, and Variable Length with Overlap Relationship Methods. All systems can only identify intrusions resulting from mismatches. If the system is not fully trained, a false positive result from signaling a normal behavior as an intrusion because it does not match an entry in the database. The systems can not identify any intrusions that do not generate any mismatches. For example, if an attack can deviate producing mismatches or the mismatch instances do not exceed the allowable mismatch threshold, then the attack will go undetected. Finally, the systems can not identify system-call-denial of service attacks. In this attack, the tested sequences exist in the normal database and go undetected if this patter is repeated indefinitely.

Third, I developed a danger theory model to detect host based intrusion detections by monitoring system call sequences. A danger theory model incorporates many mechanisms and cells to perform its functionalities. In this dissertation we have developed a model to represent such functionalities. Since each problem is domain specific and has its own requirements, we decided to adapt a simple version of danger theory model that incorporates the basic functionalities. The basic functionalities include B cell identification of bacteria, iDC identification of bacteria and signals sensing, T1 helper management of the immune system responses, T2 helper suppressing or priming B cell and T killer attacking the source of problem.

Fourth, in the enhanced system with danger theory, I instantiated one instance of each danger theory cell type. We took advantage of the danger theory functionality of an immune system and implemented it as an object oriented based system where only one instance of an object is instantiated. In general, the immune system employs millions of B or T cells to perform the same functionality by having different sensors that identify different antigens (i.e. intrusion instances). For example, a set of B cells is responsible for identifying a specific antigen signature. Due to the overhead produced when creating many instances of such entities, and the need to maintain, manage and handle signaling among them, we approached the problem in a different way. Our system is not represented as populations of autonomous agents that exist within a distributed environment similar to current systems implementing and deploying both innate and adaptive immune concepts. Rather, in our system, each cell type within the adaptive immune system is instantiated once and handles all antigen signatures. This is accomplished by associating a database of all antigen signatures that should be monitored to B cell and iDC objects. At the same time our system allows the instantiation of more than one B cell object but we reserve such a choice to the following conditions. First, if we decide to monitor more than one application, then each application may have its own B cell object associated with its appropriate database. Second, on multi processor systems, the database can be divided and each part can be associated with an instance of the B cell and they can work concurrently.

1.4. Dissertation Organization

This dissertation is organized as follows. Chapters 2, 3 and 4 are background information about related information. Chapter 2 explains intrusion detection systems and in specific process anomaly detection. Chapter 3 explores the biological immune systems which inspired this work. Chapter 4 explains artificial immune systems, implemented systems that are based on such concepts and different immune system approaches to IDS.

Chapters 5, 6 and 7 are the work carried out in our dissertation. In chapter 5 we explain our off-line investigation of intrusion detection systems that uses trails of system calls. Three systems have been implemented and tested against each other. Chapter 6 explains our developed general danger theory model. Chapter 7 explains our implementation and enhancements to lookahead-pairs method. In general, enhancing lookahead-pairs method has been performed on three stages.

Finally chapter 8 states the conclusion and future work of our dissertation.

CHAPTER 2

INTRUSION DETECTION SYSTEMS (IDS)

2.1. General View of IDSs

IDSs are software systems designed to identify and prevent the misuse of computer networks and systems. James Anderson was one of the first people to discuss IDSs [Anderson 1980] and Dorothy Denning was the first to discuss an IDS implementation [Denning 1987]. There have been attempts to classify IDS such as in the works of [Axelsson 1999; Axelsson 2000; Debar, Dacier and Wespi 2000] where an IDS is classified in two classes: misuse and anomaly detection. The misuse detection approach examines network and system activity for known misuses, usually through some form of pattern-matching algorithm. In contrast, the anomaly detection approach bases its decisions on comparing against a profile of normal network or system behavior. Any event that does not conform to this profile is considered anomalous. Both approaches have strengths and weaknesses. Misuse-based systems generally have very low false positive rates, but they are unable to identify novel attacks, which leads to high false negative rates. On the other hand, anomaly-based systems are able to detect novel attacks but produce high number of false positives. This is because current anomaly-based techniques don’t handle real world normal and legitimate computer usages that might have changed over time [Kim et al. 2007].

IDSs can also be classified according to their placement which can be as host-based, network-based or hybrid systems. Host-based systems are present on each monitored host, and collect log files of the host's operation, network traffic to and from the host, or information on processes running on the host [Kim and Spafford 1993] [Xie et al. 2004]. In contrast, network-based IDSs monitor the network traffic on the network containing the hosts to be protected, and are usually run on a separate machine termed a sensor [Leach and Tedesco 2003]. Host-based systems are able to determine if an attempted attack was indeed successful. It can also detect local attacks, privilege escalation attacks and attacks which are encrypted. However, such systems can be difficult to deploy and manage, especially as the number of hosts increase. They are also unable to detect attacks against multiple targets of the network. On the other hand, network-based systems are able to monitor a large number of hosts with relatively low deployment costs, and are able to identify attacks to and from multiple hosts. However, they are unable to detect whether an attempted attack was indeed successful, and are unable to deal with local or encrypted attacks. Therefore, hybrid systems, which incorporate host- and network-based elements, can offer the best protective capabilities to protect against attacks from multiple sources [Kim et al. 2007]. More advanced systems exist which detect high-level intrusion scenarios through correlation of multiple low-level events. They allow for the detection of non-trivial or distributed intrusions spanning multiple events and sources. They can also combine poor quality detection results from misuse and anomaly detectors to produce more reliable results. [Valdes and Skinner 2001]'s approach finds statistical similarities between alerts. [Dain and Cunningham 2001]'s approach combines alerts into attack scenarios. However, [Ning et al. 2004] states that such approaches fail to detect an intrusion if the set of reported alerts does not constitute a complete intrusion scenario.

Furthermore, IDSs can be classified according to the overall control strategy employed. For example, in a centralized IDS, data analysis is performed and controlled in a fixed number of locations independent of the number of hosts being monitored. However, in a distributed IDS, analysis is performed in a number of locations, usually on the monitored hosts themselves and control is distributed throughout the system. In a hierarchical IDS, information gathering occurs at leaf nodes and is passed to internal nodes that aggregate information. This data is then passed through internal nodes until it reaches the root node which determines if an attack has occurred and issues appropriate responses. Thus analysis is distributed over several different components; however there still exists a central controller. Centralized IDSs are not very resilient to attacks because disabling the control components renders the entire system inoperable. They are also are not very scalable or able to cope with high volume data environments due to their centralization of data analysis and processing. Hierarchical IDSs overcome scalability and processing issues because of their more efficient communication strategy and partial distribution of components. However, a distributed IDS can consume large amounts of resources on the monitored hosts, degrading the performance of the hosts to unacceptable levels if they are not carefully implemented [Twycross 2007].

2.2. Process Anomaly Detection

A process is a running instance of a program. On modern multitasking operating systems many processes can be effectively running simultaneously. A single running program executable may create several child processes by forking or threading. For example, an initial parent process acting as a web server typically starts a child process to handle individual connections as they are received. Child processes themselves may create children generating a complex process tree. The parent of this tree is the process that was created when the executable was first run. The operating system is responsible for managing the execution of these running processes and associates a process identifier (PID) with each process. This number uniquely identifies a process. When a process is started, the operating system associates to it the PID of the parent process that created it, and the user who started the process. The process is also allocated resources by the operating system such as memory, which stores the executable code and data, and file descriptors, which identify files or network sockets which belong to the process. Often, the initial goal of an attack is to gain administrator privileges on a machine granting full and free control of the system. Furthermore, there are several general classification systems for attacks [Mell et al. 2003]. One of the most frequently used is the R2L and U2R classification. If the attacker does not have an account on the system then he may try to exploit a vulnerability in a network service running on the target remote machine to gain access. This is termed a remote-to-local or R2L attack. Buffer overflow exploits are often used to subvert remote services to execute code the attacker supplies and, for example, open a remote command shell on the target machine. Sometimes, the attacked service will already be running with administrator privileges, in which the initial attack is complete. Otherwise, the attacker will have access to the machine at the same privilege level as the attacked service is running at. In this case the attacker will need to perform a privilege escalation attack, called a user-to-root or U2R attack. Often, this will involve attacking a privileged program, such as a program running with administrator privileges and subverting its execution to create a command shell with administrator privileges. After gaining unrestricted access, the attacker may install root kits to hide their presence and facilitate later access. Data can be copied to and from the machine, remote services such as file sharing and IRC daemons can be started. In the case of worms all of this can be done automatically without human intervention. In general, process anomaly detection systems are designed to detect and prevent the subversion of processes necessary in such R2L and U2R attacks [Twycross 2007].

2.2.1. System Calls

Host-based IDSs monitor running processes to detect intrusions and collect information about a running process from a variety of sources such as log files created by the process. Monitoring the behavior of a process will indicate if the process is behaving normally or has been subverted by an attack. Although log files are an obvious starting point for such systems and are commonly used, attacks may not cause any logging to take place, and so evade detection. This is why there has been a substantial amount of research into other data sources, usually collected by the operating system such as collecting system calls (syscalls). Syscalls are a low-level mechanism by which applications request system services such as peripheral I/O or memory allocation from an operating system. As a process runs it cannot usually directly access memory or hardware devices. Instead, the operating system manages these resources and provides a set of functions, called syscalls, which processes can call to access these resources. On modern Linux systems there are around 300 syscalls, accessed via wrapper functions in the libc library. At an assembly code level, when a process wants to make a system call it will load the system call number into the EAX register, and system call arguments into registers such as EBX, ECX or EDX. The process will then raise a 0x80 interrupt. This causes the process to halt execution and the operating system to execute the requested syscall. Once the syscall has been executed, the operating system places a return value in EAX and returns execution to the process. Operating systems other than Linux differ slightly in these details, for example BSD puts the syscall number in EAX and pushes the arguments onto the stack [Bovet and Cesati 2002][syscalls]. Higher-level languages provide library calls which wrap syscalls in functions such as printf.

Syscalls are a powerful data source for detecting attacks because any application that interacts with the network, file system, memory, and other hardware devices will use system calls. Most attacks that manipulate the execution of an application will need to access some of these resources and initiate a number of system calls. Therefore, it is more difficult to deceive a system call based IDS; however, monitoring syscalls is more complex and costly than reading data from a log file. Monitoring system calls may require placing hooks or stubs which increases the runtime of the monitored process, since for each syscall the monitor will spend at least a few clock ticks pushing the data it has collected to a storage buffer. There are other shortcomings from using system call based IDS. For example, permitting or denying the syscall can add additional runtime overheads. Also, processes can generate hundreds of syscalls a second, making the data load significantly higher. Incorrect replication of operating system state or other race conditions may allow syscall monitoring to be evaded [Garfinkel 2003] [Twycross 2007].

Programs such as strace [strace] intercept and log syscalls in user-space. This is very similar to tracing a program with a debugger. Strace uses the ptrace service (itself made available through a syscall) provided by many operating systems to trace the execution of a process, in this cases the monitored application. Whenever the traced process makes a system call, its execution is halted and handed back to the tracing process. Strace then logs detailed information about the syscall before allowing the traced process to resume execution. Modifying the kernel is another popular method of capturing syscall information and is used by systems such as Snare [snare] and the Linux Trace Toolkit [LTT] [Maniatty et al. 2005] [Yaghmour and Dagenais 2000]. Snare [snare] is a widely-used application for syscall logging and analysis available for a wide range of Linux, Solaris, Windows and other platforms. A kernel patch is used to record syscall information for all processes running on the monitored system. Snare takes the approach of only monitoring a certain subset of 'sensitive' syscalls which could be used to compromise security. This reduces the amount of data recorded, and decreases performance overheads on the monitored system. The recorded syscall information is collected by a user-space audit daemon, which processes the raw data and saves it in an event log. An operator then uses a graphical front end to examine the event logs for signs of intrusions [Twycross 2007].

The Janus and Ostia syscall interposition systems of Wagner et al. [Garfinkel 2003] [Garfinkel, Pfaff and Rosenblum 2004] [Goldberg et al 1996] [Janus] [Wagner 1999] sandbox an application and resemble a firewall between an application and the operating system. These systems are based on a kernel module which intercepts syscalls, and a user-space program to implement a syscall policy. They specify a policy specification syntax indicating acceptable access to file system, memory, network and other resources. Syscalls requesting resources not specified on this policy are denied. The capture and processing of kernel-space is faster than user-space methods since it reduces overhead due to the switch from kernel to user space. Ko et al. [Fraser, Badger and Feldman 1999] [GSWT] [Ko et al. 2000] have implemented the Generic Software Wrappers Toolkit, a system for UNIX and Windows platforms which integrates confinement and intrusion detection techniques. Their system allows the integration of intrusion detection techniques into the kernel to address concerns about performance and security of user-space approaches.

Tandon and Chan [Tandon and Chan 2003] [Tandon and Chan 2005] developed a representation which combines syscall arguments and sequences, and evaluate this representation using a rule-based learning classifier. They found that the addition of syscall argument information to sequences of syscalls results in better detection of attacks. Tandon et al. [Tandon, Chan and Mitra 2004] introduced the idea of a motif, which is a repeated subsequence of syscalls within a sequence, and showed how this representation can be used to improve anomaly detection performance.

2.2.2. Approaches to Process Anomaly Detection

The performance of modern computing systems has improved the computational overheads imposed by syscall monitoring and made syscalls an important data source for process anomaly detection systems. In general, IDSs use syscalls to monitor an application for signs and possibly alerting an operator. This detection may be done in real-time or offline to help audit previously gathered log files. Additionally, some real-time systems automatically take measures to actively prevent an attack from being successful. These include denying syscalls identified as suspicious or delaying execution of the monitored application. IDSs which actively respond to an intrusion are called intrusion prevention systems [Axelsson 2000].

Ko et al. [Ko, Fink and Levitt 1994] [Ko, Ruschitzka and Levitt 1997] introduced Basic Security Module (BSM), which is the Sun Solaris audit daemon, and monitor audit logs to gather data on application syscalls. Ko et al. restrict their analysis to a subset of syscall involved with file access and program execution. Their specification-based approach describes the behavior of the permitted program by a policy specification language, and describes a specification language which allows a policy to be created specifying permissible operations on files and executables for an application. Policies can either be generated by hand [Ko, Ruschitzka and Levitt 1997] [Sekar, Bowen and Segal 1999] or by using static program analysis techniques [Wagner and Dean 2001].

Esponda et al. [Esponda, Forrest and Helman 2004] presented a formal framework for analyzing the tradeoffs between positive and negative approaches. Positive detection approaches compare current behavior against a database of permitted activity, whereas negative detection approaches compare current behavior against a database of anomalous activity [Esponda, Forrest and Helman 2004]. For small problems, they show that a positive approach is more effective than a negative one, and derive results which predict how large a problem must be in order for a negative approach to be advantageous.

Stibor [Stibor 2006] shows that certain matching approaches such as Hamming distance work poorly with negative approaches, introducing an infeasible amount of complexity. Reduction of this complexity by generalization of the matching criteria results in a significant reduction in the classification performance. Based on these observations, Stibor concludes that negative approaches such as immune-inspired negative selection are unsuitable for real-world anomaly detection problems.

The systrace system of Provos [Provos 2003] [systrace] is a syscall-based IDS for Linux, BSD and OSX systems. The kernel patch inserts various hooks into the kernel to intercept syscalls from the monitored process. The user specifies a syscall policy which is a list or database of permitted syscalls and arguments. The monitored process is wrapped by a user-space program which compares any newly generated syscalls with this policy. It then only allows the process to execute syscalls which are present on the normal list. Execution of the monitored process is halted while this decision is made, which, along with other factors such as the switch from kernel- to user-space, adds an overhead to the monitored process. However, due to the simplicity of the decision-making algorithm as well as a good balance of kernel versus user-space implementation, the performance impact on average is minimal. As an IDS, systrace can be run to either automatically deny and log all syscall attempts not permitted by the policy, or to graphically prompt a user as to whether to permit or deny the syscall. In the latter mode, a syscall can be added to the policy, adjusting it before using it in automatic mode.

Gao et al. [Gao, Reiter and Song 2004b] introduced a new model of syscall behavior called an execution graph. An execution graph is a model that is constructed from syscalls gathered during normal execution. In addition to system call number, stack return addresses are also gathered and used in construction of the execution graph. The authors also introduce a course-grain classification of syscall-based IDSs into white-box, black-box and gray-box approaches. Black-box systems build their models from a sample of normal execution using only system call number and argument information. Gray-box approaches build their models from a sample of normal execution by using also additional runtime information. White-box approaches do not use samples of normal execution, but instead use static analysis techniques to derive their models. A prototype gray-box anomaly detection system using execution graphs is introduced by the authors, and they compare this approach to other systems, and discuss possible evasion strategies in [Gao, Reiter and Song 2004a].

Sekar et al. [Sekar et al. 2001] implement a real-time IDS which uses finite state automata (FSA) to capture short and long term temporal relationships between syscalls. One advantage of using FSA to evaluate sequences of syscalls is that there is no limit to the length of the syscall sequence. Yeung et al. [Yeung and Ding 2003] described an IDS which uses a discrete hidden Markov model trained using the Baum-Welch re-estimation algorithm to detect anomalous sequences of syscalls. In [Kruegel et al. 2003], Krugel et al. describe a real-time IDS implemented using Snare under Linux. Their system automatically detects anomalies in syscall arguments. They explore a number of statistical models which are learnt from observed normal usage. Endler [Endler 1998] presents an offline IDS which examines BSM audit data. It combines a multi-layer perception neural network which detects anomalies in syscall sequences with a histogram classifier which calculates the statistical likelihood of a syscall. Lee and Xiang [Lee and Xiang 2001] evaluate the performance of syscall-based anomaly detection models built on information-theoretic measures such as entropy and information cost. They also used these models to automatically calculate parameter settings for other models.

Forrest, Hofmeyr, Somayaji and other researchers at the University of New Mexico have developed several immune-inspired learning-based approaches. Forrest et al. [Forrest et al 1996] [Forrest, Hofmeyr and Somayaji 1997] [Hofmeyr and Forrest 1998] evaluated a real-time system which detects anomalous processes by analyzing sequences of system calls. Syscalls generated by an application are grouped together into sequences. A database of normal sequences is constructed and stored as a tree during training. Sequences of syscalls are then compared to this database using a Hamming distance metric, and a sufficient number of mismatches generate an alert. No user-definable parameters are necessary, and the mismatch threshold is automatically derived from the training data. Similar approaches have also been applied by this group to network intrusion detection [Balthrop et al. 2002] [Balthrop, Forrest and Glickman 2002] [Hofmeyr 1999] [Hofmeyr and Forrest 2000] [Hofmeyr and Forrest 1999a].

Somayaji [Somayaji 2002] [Somayaji and Forrest 2000] developed the immune-inspired pH intrusion prevention system which detects and actively responds to changes in program behavior in real-time. Sequences of syscalls are gathered for all processes running on a host and compared to a normal database. If an anomaly is detected, execution of the process that produced the syscalls will be delayed for a period of time. This method of response, as opposed to more malign responses such as killing a process, is more benign in that if the system makes a mistake and delays a process which is behaving normally, this may not have a perceptible impact from the perspective of the user.

Greensmith [The Danger Project] has used Libtissue to implement an immune-inspired process anomaly detection system [Greensmith, Aickelin and Twycross 2006] [Greensmith, Twycross and Aickelin 2006]. Their algorithm, called DCA, is inspired by biological Dendritic cells (DCs). A population of artificial DCs is created and monitors a host, collecting process IDs (PIDs) of the processes currently running. These PIDs are used as an antigen and stored by the DC. DCs also monitor a number of statistics for the host, such as outgoing packet and ICMP error message rates. These statistics are used as input signals for the DCs and govern their behavior. Different signals such as safe and danger signals are weighted and combined to create output signals for each DC. Over time, if the summation of the output signals exceeds a user-defined threshold, the DC matures and is removed from the system.

CHAPTER 3

BIOLOGICAL IMMUNE SYSTEMS

The Human Immune System (HIS) or the biological immune system is a robust, complex, adaptive system that defends the body from foreign pathogens. It categorizes cells within the body as self-cells or non-self cells [Dasgupta 2004; Aickelin and Dasgupta 2005; Hofmeyr 2000]. The immune system is a multi-layered defense system that protects living organisms from disease. These layers consist of physical and chemical barriers and specialized cells that can recognize and kill antigens. The mechanical and chemical barriers such as skin, mucous secretions and enzymes with their changing pH and temperature features provide the first line of defense against antigens. Bacteria on the skin surface are generally unable to pass through the skin barriers. The second line of defense is the innate immune system and it consists of a family of cells called phagocytes that recognize, attack, and then kills antigens. The innate response is non antigen-specific and was meant to fight against any infection without the need of previous immunization. It has two different actions: rapid action which lasts from four minutes to four hours performed by macrophages. There is also a medium to slow action performed via inflammation or by natural killer (NK) cells [Pagnoni and Visconti 2005]. When the innate system fails, an infection is established and the acquired immunity starts to develop. The acquired immune response is based on a complex learning process that makes the immune system adaptively acquire better immunity during its lifetime

[Aickelin and Dasgupta 2005]. The immune system uses multilevel defense both in parallel and sequential fashion. Depending on the type of the pathogen, and the way it gets into the body, the immune system uses different response mechanisms either to neutralize the pathogenic effect or to destroy the infected cells. The human immune system features that are relevant to intrusion detection are matching, diversity and distributed control. Matching refers to the binding between antibodies and antigens. Diversity refers to achieving optimal antigen space coverage and distributed control means that there is no central controller. This process depends on two important white blood cells called T-cells and B-cells. Both originate in the bone marrow, but T-cells pass on to the thymus to mature, before circulating in the blood. The T-cells are of three types: helper T-cells which are essential to the activation of B-cells, killer T-cells which bind to foreign invaders to destroy them, and suppressor T-cells which inhibit the action of other immune cells thus preventing allergic reactions and autoimmune diseases. Finally, B-cells are responsible for the production and secretion of antibodies, which are specific proteins that bind to the antigen [Dasgupta 2004; Aickelin and Dasgupta 2005; Hofmeyr 2000].

There have been several attempts to summarize immune system mechanisms. The following are five attempts:

- Immune Network Theory: The hypothesis of the immune network theory states that the immune system maintains an idiotypic network of interconnected B-cells for antigen recognition. These cells both stimulate and suppress each other in certain ways that lead to the stabilization of the network. Two B-cells connect if their shared affinities exceed a certain threshold, and the strength of the connection is directly proportional to the affinity they share [Dasgupta and Atooh-Okine 1997; Aickelin and Dasgupta 2005].

- Negative Selection Mechanism: The purpose of negative selection is to provide tolerance for self-cells. It is concerned with the immune system's ability to detect unknown antigens while not reacting to self cells. During the generation of T-cells, receptors are made through a pseudo-random genetic rearrangement process and then undergo a censoring process in the thymus, called the negative selection. If T-cells react against self-proteins, they are destroyed allowing only the T-cells that don't react to self-proteins to leave the thymus and circulate throughout the body to perform immunological functions and protect the body against foreign antigens [Dasgupta and Atooh-Okine 1997; Aickelin 2004; Aickelin and Dasgupta 2005].
- Clonal Selection Principle [Aickelin 2004; Aickelin and Dasgupta 2005] describes the basic features of an immune response to an antigenic stimulus. Only the cells that recognize the antigen proliferate and are selected against those that do not.
- Idiotypic Networks—Network Interactions (Suppression): The idiotypic network hypothesis [Aickelin and Dasgupta 2005; Cayzer and Aickelin 2002b; Cayzer and Aickelin 2005] builds on the recognition that antibodies can match other antibodies as well as antigens. This could be used to explain how the memory of past infections is maintained and could result in the suppression of similar antibodies and encouraging diversity in the antibody

pool. In general, the nature of an idiotypic interaction can be either positive or negative and it's matching function symmetric.

- Danger Theory: The proposed Danger Theory [Matzinger 2002; Aickelin and Dasgupta 2005] is assumed to provide a method of "grounding" the immune response. The Danger Theory states that there must be discrimination happening other than the self–non-self distinction. Danger theory discriminates "some self from some non-self" or "non-self but harmless" and of "self but harmful". The central idea in the Danger Theory is that the immune system does not respond to non-self but to danger. In this theory, danger is measured by damage to cells indicated by distress signals that are sent out when cells die an unnatural death, as opposed to programmed cell death. Essentially, the danger signal establishes a danger zone, as shown in Figure 3.1, around itself. The B-cells producing antibodies that match antigens within the danger zone get stimulated and undergo the clonal expansion process. Those that do not match or are too far away do not get stimulated. In general, the danger signal can be a "positive" signal or a "negative" signal. [Matzinger 1994, Matzinger 2002] proposed the Danger model, which suggests that the immune system is more concerned with damage than with foreignness, and is called into action by alarm signals from injured tissues, rather than by the recognition of non-self only. The danger theory proposed that APCs are activated by danger/alarm signals from injured cells, such as those exposed to pathogens, toxins, mechanical damage, and so forth.

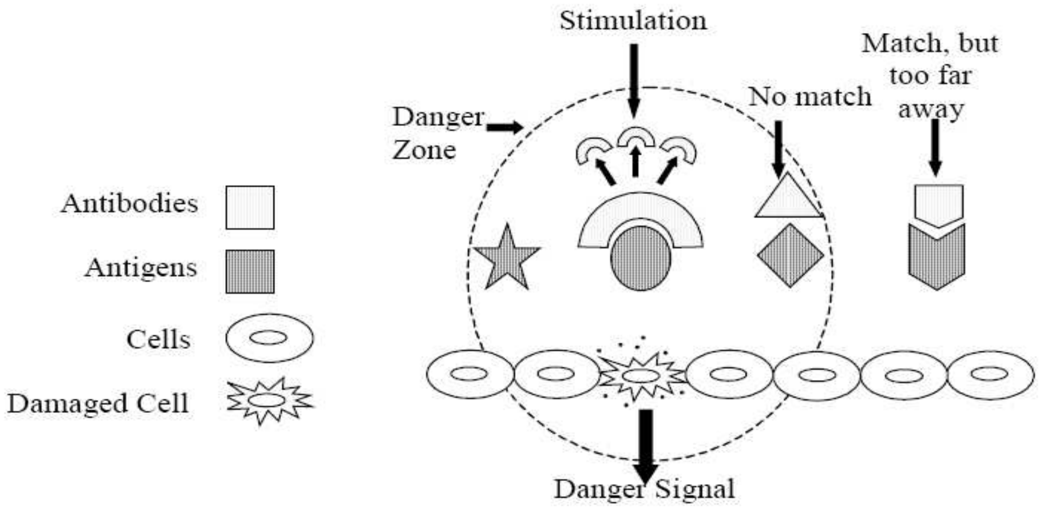


Figure 3.1. Danger theory illustration [Aickelin and Dasgupta 2005]

As shown in Figure 3.2., a B cell receives signal 1 from bacteria and sends signal 1 to a T-helper cell (Th). At the same time Antigen Presenting Cell (APC) receives signal 0 from both the bacteria and distressed cell. This signal is transformed to signal 2 which is sent to Th along with signal 1 from APC which recognized a foreign body. Then Th sends signal 2 to both B cell and other T-killer cells (Tk). At the same time Tk could have received signal 1 from cell infected by a virus.

Structurally, the immune system is a collection of cells, molecules, tissue, organs and circulatory systems [Jeneway et al. 2005]. Immune system cells are produced and mature in specialized areas of the body called primary lymphoid organs such as the thymus or bone marrow. They are transported via the cardiovascular and lymphatic circulatory systems to peripheral tissues or specialized secondary lymphoid organs such as the lymph nodes or spleen. Microorganisms attempt to consume the body. Damage to the body is called pathology, and the damaging agent, such as bacteria or virus, a pathogen. Functionally, the human immune system is able to locate and remove many of these pathogens from the body and maintain the body in a healthy state for many years.

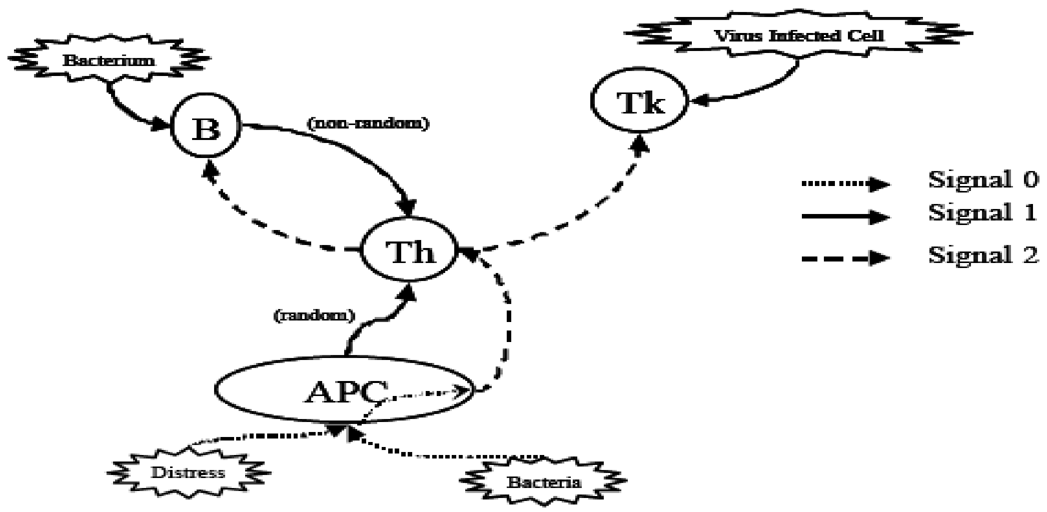


Figure 3.2. Danger theory viewed as immune signals [Matzinger 1994]

CHAPTER 4

ARTIFICIAL IMMUNE SYSTEMS (AIS)

4.1. Introduction

Differentiating between normal and intrusive activities is one of the major challenges facing computer security. AIS, which is a biologically inspired computing, is currently investigated to solve this problem. Such a method was inspired by the Human Immune System (HIS) that can detect and defend against harmful and previously unseen invaders. An analogy can be drawn between the HIS and IDS. The innate part of the HIS is similar to the misuse detector class of IDS whereas the adaptive immune system is closer to an anomaly based IDS. Both the innate HIS and misuse detectors have prior knowledge of attackers and detect them based on this knowledge. Both the adaptive immune system and anomaly detectors generate new detectors to find previously unknown attackers [Kim et al. 2007]. HIS protects the body against damage from an extremely large number of harmful bacteria, viruses, parasites and fungi, termed pathogens. It does this usually without prior knowledge of the structure of these pathogens. This property, along with being distributed, self-organized and lightweight [Kim 2002] made HIS the focus of computer science and intrusion detection communities. This is because it can be viewed as a form of anomaly detector with very low false positive and false negative rates. AISs have been built for a wide range of application domains including document classification, fraud detection, and network- and host-based intrusion detection. In specific AIS approaches for intrusion detection has been reviewed by Aickelin et al. [Aickelin, Greensmith and Twycross 2004] AISs can be broadly divided into two categories based on the mechanism they implement: network-based models and population-based models with the existence of many hybrid models. Network based models are based on Jerne's idiotypic network theory which recognizes interactions between antibodies and antibodies as well as between antibodies and antigens. Population-based models use negative or clonal selection as the method of generating and maintaining a population of detectors [Twycross 2007].

4.2. Artificial Immune Systems Basic Concepts

To implement a basic artificial immune system, four decisions have to be made: encoding, similarity measure, selection and mutation. After fixing a suitable encoding and choosing a suitable similarity measure, the algorithm will then perform selection and mutation; both based on the similarity measure, until the stopping criteria are met.

4.2.1. Initialization/Encoding

It is very important to choose a suitable encoding [Aickelin 2004; Aickelin and Dasgupta 2005] for the algorithm's success. In order to perform encoding, the antigen and antibody should be defined in the context of an application domain. Antigens represent intrusion data instances. Antibodies bind to antigen identifying an intrusion. Sometimes there can be more than one antigen at a time and there are usually a large number of antibodies present simultaneously. Both antigens and antibodies are represented or encoded in the same way.

4.2.2. Similarity or Affinity Measure

It is very important to choose a good matching algorithm for the artificial immune system to work properly. The primary response in the immune system [Forrest and Hofmeyr 2001a] uses learning mechanism for new antigens that have not been detected by a detector before. When a B cell is activated after binding to a pathogen, it starts cloning itself and the cloned cells then undergoes a somatic hyper mutation to create daughter B cells with mutated receptors and then the new B cells will compete with their parents. In general, the higher the affinity of B cell for available pathogens the more likely it will be cloned resulting in a variation and selection process called affinity maturation.

4.2.3. Negative Selection

One of the common techniques used is the negative selection algorithm [Aickelin 2004; Aickelin and Dasgupta 2005] where a set of trusted behavior “self” is defined. During the initialization of the algorithm, a large number of detectors (strings similar to intrusion instances) are created. Then these detectors are subjected to a matching algorithm that compares them to “self”. Any matching detector would be eliminated and those that do not match are selected (negative selection). All non-matching detectors will then form the final detector set. This detector set is then used in the second phase of the algorithm to continuously monitor all network traffic. In case a match occurs, this will be reported as a possible alert or “non-self”.

4.2.4. Somatic Hyper mutation

Somatic hyper mutation [Aickelin 2004; Aickelin and Dasgupta 2005] is an optional process and associated with negative selection. Rather than ignoring matching detectors in the first phase of the algorithm, they can be mutated to save time and effort. Also, depending on the degree of matching, the mutation could be more or less strong. [Forrest and Hofmeyr 2001b] use, in their immune system, permutation masks to achieve diversity similar to the role of the major histo-compatibility complex (MHC). MHC is responsible for transporting peptides from the interior regions of a cell and presents it on its surface. A permutation mask defines a permutation of the bits in the string representation of the network packets. In general, each detector has a unique randomly generated permutation mask.

4.2.5. Cross-Reactivity and Associate Memories

When a B-cell encounters subsequent antigens it responds quicker (secondary response) in which the memory cells for the earlier antigen quickly start producing large quantities of a specific antibody. In general, B-cell receptors do not require an exact match to an antigen to be activated. Therefore, some memory cells can react to new antigen producing a secondary response which is termed, the cross-reactive memory [Forrest and Hofmeyr 2001b]

4.3. Artificial Immune System Applications

This section briefly introduces some application areas where AIS have been applied.

4.3.1 Virus Detection

Since computer viruses have been identified as a destructive form of artificial life it is very natural for computer scientists to investigate the human immune system in order to understand its defense mechanism against harmful biological viruses. Virus detection is viewed as a self-non-self discrimination problem. Targets such as legal user activities, legal application usage activities, and uncorrupted data are monitored as self and the AIS are expected to discriminate them from illegal user activities, illegal application usage activities, and virus infected data. In general, detectors are generated from a standard binary executable .com file and then the generated detectors are checked to see if they can detect a virus infected .com file. Another recent approach called Computer Virus Immune System (CVIS) employs the negative selection algorithm with some novel ideas that were used in [Hofmeyr 1999; Hofmeyr and Forrest 2000] such as life span, activation threshold and costimulation. This new technique performs virus analysis, repairs infected files, analyze the results of other local systems and operates under a distributed environment using autonomous agents. A different approach to using AIS for virus detection is undertaken at the IBM Research Centre. They attempt to identify and understand useful processes of the human immune system, and to see how these can help in developing a new virus detection product. However, they do not attempt to implement the processes using the mechanism of the human immune system, only to mimic it at a high level of abstraction [Kim 2002].

4.3.2. Recommender Systems

Collaborative filtering (CF) [Cayzer and Aickelin 2002a; Chao and Forrest 2002] is one of the common applications of AIS. CF is the term for a broad range of algorithms that use similarity measures to obtain recommendations. In general, any problem domain where users are required to rate items is amenable to CF techniques. Usually, commercial applications are called recommender systems one of which is movie recommendation. Traditionally, recommended items are treated as ‘black boxes’ and recommendations are based purely on the votes of neighbors, and not on the content of the item. The preferences of a user, usually a set of votes on an item, comprise a user profile, and these profiles are compared to build a neighborhood. Data encoding where a user profile is presented as a string of numbers and the similarity measure which is usually a correlation-based measure are the key decisions to be made.

[Morrison and Aickelin 2002] applied idiotypic network theory to build their web site recommender AIS based system. The idiotypic network theory states that interaction in the immune system do not only occur between antibodies and antigens but also between antibodies and each other. Therefore, the antibody may be matched by other antibodies. This activation can continue to spread throughout the population. This interaction may have a positive or a negative effect on a particular antibody-producing cell. This theory, therefore, can explain how the memory of past infections is maintained and that could result in the suppression of similar antibodies thus encouraging diversity in the antibody pool. Morrison and Aickelin idea' is that antibodies that are very similar to each other had their concentrations reduced. This allowed the creation of a set of users that are similar to the user but quite different to each other and thus enhancing the recommendation accuracy of the system.

4.3.3. Intrusion Detection

The use of artificial immune system in intrusion detection is beneficial because immune systems can provide a high level of protection from invading pathogens in a robust, self-organized and distributed manner and is capable of coping with the dynamic and complex nature of computer system security [Aickelin, Greensmith and Twycross 2004]. Human immune systems (HIS) can detect and defend against harmful and previously undetected pathogens and has the properties of being error tolerance, adaptive and self-monitoring. The HIS system protects the body from pathogens without any prior knowledge of their structure making the system distributed, self-organized and lightweight. The HIS is also seen as a form of anomaly detector with low false positive and false negative rates.

4.4. AIS Features and Principles for IDS

[Somayaji, Hofmeyr and Forrest 1998; Hofmeyr 1999] [Kim 2002] [Somayaji 2002] presented several immune features that are desirable for an effective IDS and identified the following principles that will guide the process of building an intrusion detection system based on immune system concepts:

- Distributed protection: Lymphocytes in the immune system determine locally the presence of an infection with no central coordination taking place.
- Scalability: the immune system is scalable since communication and interaction between components are localized and there is little overhead associated when the number of components is increased.
- Multi-layered: In the immune system, security is achieved by combining multiple layers of different mechanisms to provide high overall security.
- Diversity: diversity ensures that security vulnerabilities in one system are less likely to be widespread.
- Robustness or Disposability: No single component or cell of the human immune system is essential and can be replaced.
- Autonomy: The immune system does not require outside management or maintenance as it classifies and eliminates pathogens, and it repairs itself by replacing damaged cells.

- Adaptability: The immune system learns to detect new pathogens, and retains the ability to recognize previously seen pathogens through immune memory.
- No secure layer: Any cell in the human body can be attacked by a pathogen including those of the immune system itself. However, because lymphocytes are also cells, they can protect the body against other compromised lymphocytes.
- Dynamically changing coverage: since the immune system cannot maintain a set of detectors large enough to cover the space of all pathogens, it maintains a random sample of its detector repertoire circulating throughout the body.
- Identity via behavior: In cryptography, identity is proven through the use of a secret. The human immune system, in contrast, does not depend on secrets; instead, identity is verified through the presentation of peptides, or protein fragments.
- Anomaly detection: The immune system has the ability to detect pathogens that it has never encountered before thus performing anomaly detection.
- Flexibility or Imperfect detection: By accepting imperfect detection, the immune system increases the flexibility with which it can allocate resources.
- Detector replication: The human immune system replicates detectors to deal with replicating pathogens.
- Memory (signature based detection): Adaptation of an organism remains throughout its life time. Memory allows the immune system to react more rapidly the second time against pathogens that are similar to the ones that were encountered previously which is similar to signature based detection.

• Implicit policy specification: definition of self in immune system is empirically defined by monitoring proteins that are currently in the body. Self is defined as the actual normal behavior and not what it should be by defining it in a security policy.

4.5. Conceptual Frameworks for AISs

Stepney et al. [Stepney et al. 2005] developed a conceptual framework within which biologically-inspired models and algorithms can be developed and analyzed. In it the probes provide the experimenter with an incomplete and biased view of a complex biological system which allows the construction and validation of biologically-inspired algorithms. This is achieved by simplifying the abstract representations analytical computational frameworks. Stepney et al. also developed a meta-framework which allows common underlying properties of classes of models to be analyzed by asking questions, called meta-probes, of each of the models under consideration. Using this meta-framework, the authors analyze the commonalities of population and network models.

Neal and Timmis [Neal and Timmis 2005] present a conceptual framework which integrates artificial neural networks, AISs and artificial endocrine systems in a biologically-realistic way. The view the biological organism as a homeostatic system with self-organization is the driving force behind this homeostasis. Each of the neural, immune and endocrine systems interacts to achieve homeostasis. The biological immune system is primarily concerned with self-assertion. In their model, the artificial immune and endocrine systems control the artificial neural network. Their AIS, modeled as an idiotypic immune network, removes cells that have a negative impact on the system.

4.6. Immune System Approaches to IDS

[Kim et al. 2007] indicated that applying immune system concepts or approaches to IDS have the following major roots and distinct philosophies:

1. Methods inspired by the immune system that employ conventional algorithms, for example, IBM’s virus detector [Kephart 1994].
2. The negative selection paradigm as introduced by Forrest [Somayaji 2002] [Forrest et al. 1994].
3. Approaches that exploit the Danger Theory [Matzinger 1994].
4. Other algorithms.

4.6.1. Conventional Algorithms in AIS

[Kephart 1994; Kephart et al. 1998] designed their AIS with five major stages all inspired by the HIS. For example, the first stage detected a previously unknown virus on a user's computer which is similar to the innate human immune system. This was carried out using generic techniques and neural networks, which were used to build a generic classifier. Their proposed system first detected viruses using either fuzzy matching from a pre-existing signature of viruses, or through the use of integrity monitors which monitored key system binaries and data files for changes. In order to decrease the potential for false positives in the system, if a suspected virus was detected it was decoyed by the system to infect a set of decoy programs whose sole function was to become infected. Then a proprietary algorithm was used to automatically extract a signature for the program and then sent to the neighboring systems and the infected binaries were cleaned.

[De Paula, de Castro and de Geus 2004] proposed another AIS based IDS called ADENOIDS. They introduced eight different components taken from the innate and the adaptive immune system. From the innate immune system, the evidence-based detector is responsible for detecting intrusions based on clear evidence such as a security policy violation. The innate response agent reacts to attacks detected by the evidence-based detector. The response was to limit bandwidth or disk access. The behavior-based detector, which is an anomaly detector, is initiated only when it receives co-stimulation signals. Similar to the adaptive immune system, the signature extractor extracts signatures of detected attacks and has a learning mechanism which allows attack signatures to mature. Some of the matured attack signatures are kept at the knowledge-based detector which corresponds to the adaptive immune memory. The signature extractor activates the response generator and the adaptive response agent. The response generator decides on the response type and the adaptive response agent performs the selected responses.

4.6.2. Negative Selection (NS)

Negative selection concepts are concerned with eliminating immature cells that bind to self antigens. This allows the HIS to detect non-self antigens without mistakenly detecting self-antigens.

[Smith, Forrest and Perelson 1993][Forrest et al. 1994] proposed algorithms consisting of three phases: defining self, generating detectors and monitoring the occurrence of anomalies. In the first phase, self cells are defined by regarding normal pattern profiles as self patterns. In the second phase, a number of random patterns are generated and then compared to each of the self-patterns defined in the first phase. If any randomly generated pattern matches a self-pattern, this pattern is removed; otherwise, it becomes a detector pattern and monitors the system's profiled patterns. During the monitoring stage, if a detector pattern matches any newly profiled pattern it is considered an anomaly.

Forrest et al. [Forrest et al. 1994] [Forrest et al. 1996] viewed virus detection as a self-non-self discrimination problem within a computer. They regarded monitoring targets (such as legal user activities, legal application usage activities, uncorrupted data, etc.) as self and expected the NS algorithm to discriminate them from others (such as illegal user activities, illegal application usage activities, virus infected data, etc.). They randomly generated binary string detectors and selected the subset which did not match to self strings from a standard binary executable .com file. The experimental results showed that the NS algorithm obtained a 100% detection rate under a relatively small scale problem: with 125 detectors when an infected file was encoded by 655 binary strings each string having 32 bits.

The process of generating the repertoire is shown in Figure 4.1. using an r-contiguous matching rule and r = 2, the string to be protected is logically segmented into four equal- length \self" strings (stored in S). To generate the repertoire, random strings are produced in the box labeled R_0 and matched against each of the self strings. The first two strings, 1000 and 1100, are eliminated because they both match self string 0000 at least two contiguous positions. The string 1101 fails to match any string in self at least two contiguous positions, so it is accepted into the repertoire (box labeled "R"). Figure 4.2. explains the process of monitoring protected strings for changes.

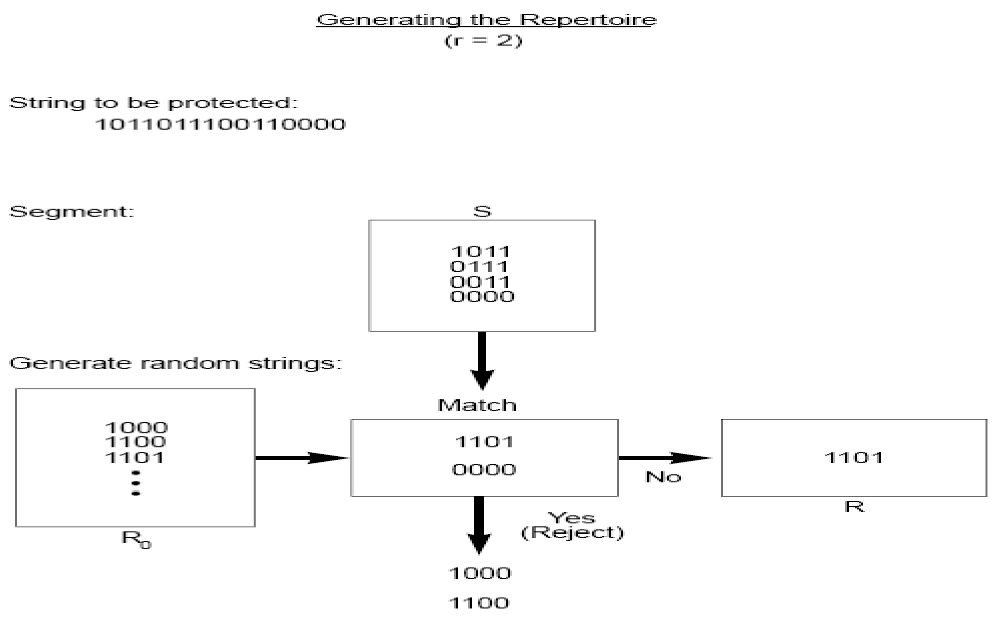


Figure 4.1. Generating the repertoire [Forrest et al. 1994].

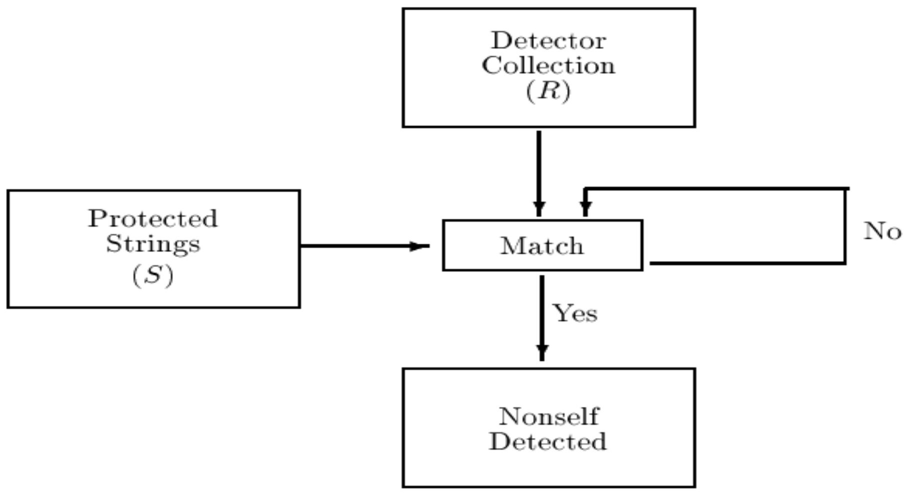


Figure 4.2. Monitor Protected Strings for Changes. [Forrest et al. 1994]

[Hofmeyr 1999; Hofmeyr and Forrest 1999a; Forrest and Hofmeyr 2001a] developed an AIS that is based on the negative selection technique. The life cycle of a detector is shown in Figure 4.3. and starts by having the detector randomly created and then remains immature for a certain period of time, which is the tolerization period.

If the detector matches any sting a single time during tolerization, it is replaced by a new randomly generated detector string. If a detector survives immaturity, it will exist for a finite lifetime. At the end of that lifetime it is replaced by a new random detector string, unless it has exceeded its match threshold and becomes a memory detector. If the activation threshold is exceeded for a mature detector, it is activated. If an activated detector does not receive costimulation, it dies (the implicit assumption is that its activation was a false positive). However, if the activated detector receives costimulation, it enters the competition to become a memory detector with an indefinite lifespan. Memory detectors need only match once to become activated.

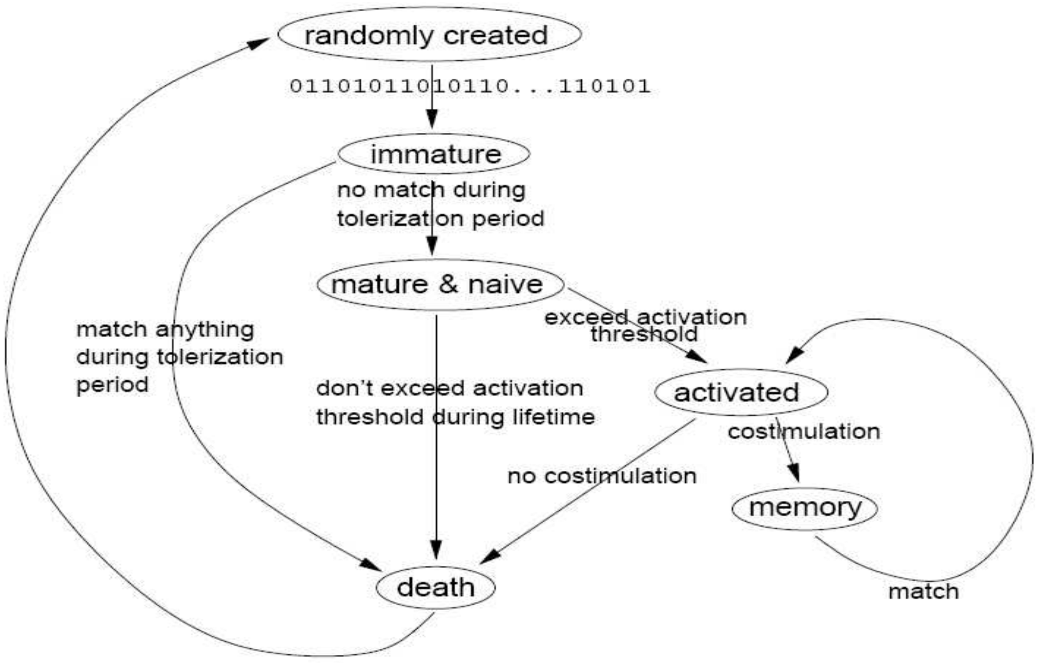

randomly created
0110101101011 0...110101
immature
no match during tolerization period
mature & naive
exceed activation threshold
match anything during tolerization period
don't exceed activation threshold during lifetime
activated
costimulation
no costimulation
memory
death
match

Figure 4.3. Life cycle of a detector [Hofmeyr 1999; Hofmeyr and Forrest 1999a; Forrest and Hofmeyr 2001a].

[Hofmeyr and Forrest 1999b] employed the use of permutation masks to increase the effectiveness of negative detection. They employed activation thresholds to allow the system to aggregate foreign activity over time, and they used adaptive thresholds that allow the system to integrate foreign patterns from multiple locations. In general, the work of Hofmeyr and Forrest [Hofmeyr 1999] [Hofmeyr and Forrest 1999a] involved the development of an AIS for network intrusion detection, called LYSIS. LYSIS implements the AIS architecture called ARTIS described in [Hofmeyr and Forrest 2000]. It employs the NS algorithm for binary detector generation and various features of the HIS such as activation threshold, life span, memory detectors, costimulation, tolerization period and a decay rate to monitor self and non-self. LYSIS is network-based and examines TCP connections, classifying normal connections as self, and everything else as non-self. This is achieved by extracting a data path triple consisting of <source host IP address, destination host IP address, TCP service (port) number from TCIP/IP packet headers>. This data path is used as input data to build self-profiles. Detectors in the form of binary strings which do not match to self-profiles for a tolerization period are generated using NS. These detectors are then used to match sniffed triplets from the network using an r-contiguous bit matching scheme. In general, r-contiguous matching measures the similarity between two binary strings by counting contiguously matching bits. If a detector matches a number of strings above an activation threshold, an alarm is raised. Detectors that produce many alarms are promoted to memory cells with a lower activation threshold to form a secondary response system. Generated detectors monitor a network for their life span periods. Co-stimulation is provided by a user confirming if an alert is actually an intrusion attempt.

[Kim and Bentley 1999d] [Kim and Bentley 1999b] proposed system consists of a primary IDS and secondary IDSs. The primary IDS are equivalent to the bone marrow and thymus and generate numerous detector sets. Each individual detector set describes abnormal patterns of network traffic packets. Local hosts are considered as secondary lymph nodes, detectors as antibodies and network intrusions as antigens. At the secondary IDS's, detectors are background processes which monitor whether non-self network traffic patterns are present. There are three evolutionary stages: gene library evolution, negative selection and clonal selection. During the negative selection stage, the system generates diverse pre-detector patterns and selects mature detector patterns by eliminating false pre-detector patterns by binding them to self patterns. Pre-detectors are generated from a gene library containing various genes. To resolve the excessive computational time caused from the random generation approach applied in negative selection [Kim and Bentley 1999a] adapted the niching strategy to build a valid detector set. The modified negative selection algorithm with niching simply replaces the random generation of pre-detectors with the evolution of pre-detectors towards 'non-self'. In the first phase, the modified negative selection algorithm builds self profiles. Then, the profiles are encoded in an appropriate data representation. In the second phase, the negative selection algorithm with niching starts generating detectors. This second phase is repeated for each self profile until all the self profiles have their own detector sets. In the third phase, the detector patterns in each detector set are compared to the new self profile. If the similarity between any detector pattern and new self pattern is beyond a predefined threshold, the algorithm generates an alarm signal.

In order to investigate the feasibility of the NS algorithm in a real network environment, [Kim and Bentley 2001a; Kim 2002] studied the problem of scalability of the NS algorithm. For this study, they used TCP packet headers covering around 20 minutes and containing five specified attacks. A total of 33 different attributes were extracted describing a specific network connection. These attributes contained the following information: connection identifier, known port vulnerabilities, 3-way handshake details and traffic intensity. For detector matching, the r-contiguous matching method was used. Non-self detection rates for the various attacks were recorded as less than 16% so the detector coverage in this case was not sufficient. It was estimated that for an 80% detection rate it would take 1,429 years to produce a detector set large enough to achieve this kind of accuracy, using just 20 minutes worth of data, and 6×10^8 detectors would be needed. From these results, the authors concluded that the NS algorithm produced poor performance due to scaling issues on real-world problems.

[Kim and Bentley 2002c] introduced dynamic clonal selection algorithm DynamiCS which starts by seeding initial immature detectors with random genotypes. DynamiCS then employs negative selection by comparing immature detectors to the given antigen set. As the result, immature detectors that bind to any antigens are deleted from the immature detector population and new immature detectors are generated until it reaches the maximum size of the non-memory detector population. In their experiments, three important parameters: tolerization period, activation threshold and life span were tested. The system performance measured by true positives (TP) and false positives (FP) rates was primarily controlled by the number of detector activations in total, and that this number was directed by values of the three parameters. A large tolerization period directly lowered FP by allowing more immature detectors to remain and pushing mature detectors out. It was also found that both lowering the activation threshold and increasing life span could guide the system to attain a higher TP rate. From analysis, lowering A and increasing L should be considered together in order to obtain an effective application of DynamiCS.

[Kim and Bentley 2002b] extended DynamiCS, so that it can handle memory detectors based on their detection results. The experiment results indicated the important role of memory detectors. They indeed contribute to increase TP rates by detecting re-encountering antigens. Without memory detectors, TP rates of DynamiCS fluctuate irregularly within an unsatisfying range (between 0.1 and 0.8). To overcome this problem Kim and extended DynamiCS to delete harmful memory detectors by applying costimulation to memory detectors similar to activating mature detectors.

In [Kim and Bentley 2002a], the authors tried to overcome the problem of requiring a large number of memory detector co-stimulation in order to obtain satisfactory TP rates. The system continued to maintain three detector populations: immature, mature and memory detector populations and treats a portion of the memory detector population as a gene library. In order to let memory detectors evolve towards existing non-self antigens without binding self antigens, the extended DynamiCS uses hyper-mutation in a way to generate new detectors more tuned to target non-self antigen detection. The test results of such extension achieved high TP rates without increasing the amount of co-stimulation. The test results, also, confirmed that hyper-mutation enabled the evolution of the virtual gene library and thus produced immature detectors that were better tuned to cover existing non-self antigens.

[Balthrop, Forrest and Clickman 2002; Balthrop et al. 2002] provided an in-depth analysis of the LISYS immune-based IDS. In this analysis, the adaptive mechanisms of the LISYS immune-based IDS were examined with respect to machine-learning (ML) counterparts, and the contribution of each individual component was quantified. Data was collected from an internal restricted network of computers controlled by the authors. After a week of normal activity, several attacks were performed and LISYS was able to successfully identify them. In general, activation thresholds and sensitivity levels contributed to reduce false positives and the incorporation of r-chunks and permutation masking also reduced false positives and increased true positives.

Furthermore, [Balthrop, Forrest and Glickman 2002] introduced an improvement to r-contiguous matching called an r-chunk scheme. In this scheme, only r contiguous bits of the whole detector are specified (known as the window), with the remaining becoming wild-cards and thus the partial matching is performed. However, [Esponda, Forrest and Helman 2004] reported that the r-chunk matching shows linear-time complexity against the number of self-patterns and windows but requires more space compared to the original NS algorithm. Furthermore, [Stibor et al 2005; Stibor, Timmis and Eckert 2005] has shown that the generated detector set under fits exponentially for small value r. Under fitting behavior leads a user to set the matching threshold value r near l. However, this verifies that the detector generation using the negative selection with r-chunk matching is infeasible since all the proposed variants of the negative selection algorithm have a runtime complexity which is exponential in r.

The Computer Virus Immune System (CVIS) approach [Harmer et al. 2002] is able to perform virus analysis, repair infected files and propagate the analysis results to other local systems. In addition, CVIS was designed to operate under a distributed environment using autonomous agents. They tested the TIMID virus, which infects .com files only within a local directory. The test reports showed the sensitivity of detection and error results on different matching thresholds. It showed a detection rate of up to 89% but had a very high scalability problem since it required approximately 1.05 years for generated antibodies to scan an 8GB hard disk drive. They employed some novel ideas such as life span, activation threshold and co-stimulation. The test results showed that the system was able to detect simulated intrusions without serious self detection errors. The results also verified that the co-stimulation and affinity maturation help reducing both FP and FN error rates. However, it was found that the affinity maturation required far too much computation time to be applied to the second, larger, data set. They also indicated that the high detection rates with low error rates might have been obtained because the simulated intrusions were limited.

[Le Boudec and Sarafijanovic 2003][Le Boudec and Sarafijanovic 2004][Sarafijanovic and Le Boudec 2003][Sarafijanovic and Le Boudec 2005]] built an immune-based system to detect misbehaving nodes in a mobile ad-hoc network. The authors considered a node to be functioning correctly if it adhered to the rules laid down by the Dynamic Source Routing (DSR) protocol. Each node in the network monitored its neighboring nodes and collected one DSR protocol trace per monitored neighbor. Four sequences of DSR protocol events were sampled over fixed, discrete time intervals to create a series of data sets. This created a binary antigenic representation in which each of the four genes recorded the frequency of their four sequences of protocol events. The NS algorithm was used to eliminate any antibodies which match normal behavior. Once a mature set of detectors had been generated, these antibodies were used to monitor further traffic from the node and, if they matched antigens from the node, it was classified as suspicious.

[Ayara et al. 2002] modified the original NS algorithm to use somatic hyper mutation. Somatic hyper mutation is the occurrence of a high level of mutation in the variable regions of B cells with the possible purpose of increasing the binding affinity to antigens. This new algorithm was called negative selection mutation (NSM) and performed a guided mutation on the detector which matched self data during the detector generation process. The specific parts of a detector used to match the bits to a self-string were targeted for mutation. The mutation rate was dynamically set according to the affinity between a detector and a self string: the greater the affinity, the higher the mutation rate. The number of mutations performed on the same candidate detector was restricted. The authors compared the NSM with the exhaustive NS through the tests performed on randomly generated 8-bit self data. The results illustrated that the two algorithms showed similar time complexity and detection rates with no statistical significant differences. However, the authors argued that these results were likely to be caused by the nature of randomly generated self data. This was because the executed mutations resulted in the detectors being pushed towards or away from a self-string with an equal probability.

[Gonzalez and Cannady 2004; Eiben, Hinterding, and Michalewicz 1999] improved the NSM algorithm by adopting the self-adaptive strategy of evolutionary algorithms to control the mutation rate. This strategy determines a mutation rate at every generation by selecting the standard deviation from the fittest detectors selected via a tournament selection, multiplied by Gaussian noise. A comparison with the NSM algorithm showed that the new algorithm performed better with respect to higher detection rates, lower false detection rates, and computation time taken.

[D'haeseleer, Forrest and Helman 1996; D'haeseleer 1996] discussed the problem of holes when using the NS algorithm. Depending on matching methods and strings used in the NS algorithm, there exist non-self strings called holes that are not covered by a complete detector repertoire as shown in Figure 4.4.

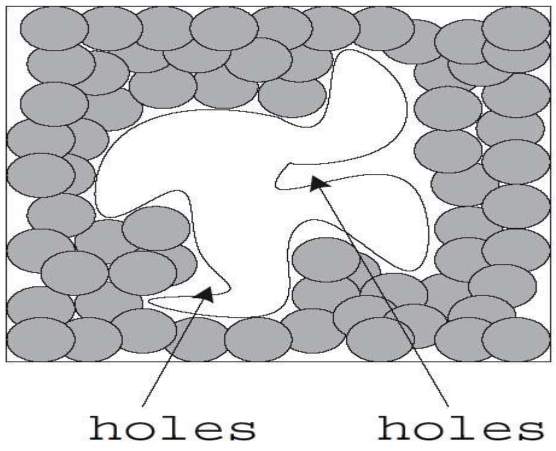


Figure 4.4. The existence of holes. Each dark circle represents a detector and a gray shape in the middle is self-antigen data. The size of the dark circles reflects the generality of detectors. Since all the detectors have an identical radii, and the detectors are too general to match some non-self subspaces without matching self antigen data, there inevitably exist holes [Hofmeyr 1999].

Such a problem results from adapting a symmetrical string matching and its generality. The existence of holes determines a lower bound on a false negative error rate. To overcome this problem, [Hofmeyr 1999; Hofmeyr and Forrest 1999b] explained that the permutation mask lets the NS algorithm randomly permutated the binary bits of generated detectors. As a consequence, it has an additional set of detectors with different representations reflecting an identical non-self space. Different representations would have different holes in a non-self space and hence the union of coverage of non-self spaces by multiple sets of detectors is likely to reduce the number of holes. The permutation mask demonstrated improved detection results by up to a factor of 3, especially when LISYS attempted to detect a non-self string close to a self-string in a search space. [Balthrop et al. 2002] investigated the effect of the permutation mask used by the simplified version of LISYS but when employing r-chunk matching. They found that the incorporation of r-chunks and permutation masking reduced false positives and increased true positives. Additionally, they found that varying r had little effect, unlike with full-length detectors. As the r-chunks scheme performed remarkably well the authors investigated it further, and subsequently found that the dramatic increase in performance was in part due to the configuration of their test network.

[Esponda and Forrest 2002] introduced positive detection as the scheme of detecting valid patterns while negative detection is the scheme detecting invalid patterns. They presented the r-contiguous bits match rule which allows both detection schemes to exhibit the same generalization. Such a rule exhibits a reduced number of holes and is able to better characterize what holes are. They concluded that negative detection is more suitable for a distributed environment. [Esponda, Forrest and Helman 2003; Esponda, Forrest and Helman 2004] showed that r-contiguous matching and permutation mask is able to cover a larger space that would be recognized by Hamming distance matching. Their study also showed that there are still non-self strings not detected by r-contiguous matching augmented by the permutation mask. They introduced crossover closure which occurs when all the possible sliding windows of each string, existing in a universal string set, exactly match the corresponding windows of some self-strings. The authors used this property to characterize two matching methods: r-contiguous and r-chunk matching. They concluded that both matching rules did not recognize all string sets under crossover closure. As a result they estimated how many self-strings are required for either negative or positive detection and approximated the number of holes as a function of self-strings coupled with a string length and a size of r. They noticed that the number of holes decreases as more self strings are added length. [Dasgupta and Gonzalez 2002; Gonzalez and Dasgupta 2002; Gonzalez 2003]. They compared a negative characterization approach to positive characterization. The positive approach focused on generating rules covering the self-space and detected anomalies by monitoring events that matched no self rules. Their implementation of the positive selection algorithm used a k-dimensional tree, giving a quick nearest neighbor search. On the other hand, their negative characterization approach employed a genetic algorithm in order to generate detector rules covering niches of a non-self space. In order for detector rules to evolve, the fitness function was defined by the volume of non-self space covered by detector rules after a penalty have been applied according to the number of matching self examples. The best detection rates they found were 95% and 85% for positive and negative selection respectively. They concluded that it is possible to use NS for IDS and that in their time series analysis, the choice of time window was important.

[Gonzalez 2003][Gomez, Gonzalez and Dasgupta 2003][Gonzalez and Dasgupta 2003] extended the negative approach to generate detectors by employing fuzzy rules. They also provided better definition of the boundary between a self and a non-self space and were able to show improved detection accuracy because of the reduction of a search space due to the fuzzy representation. [Gonzalez, Dasgupta and Kozma 2002] also developed the real-valued negative selection (RNS) algorithm. The RNS algorithm employs two distinctive features: the use of real-value representation and hybridizing the NS algorithm with a classifier. The RNS algorithm uses n-dimensional vectors as detectors. Detectors have a radius r, representing hyper-spheres in combinations with a fuzzy Euclidean matching function. In training, detectors are generated randomly and then moved to both maximize the coverage of a non-self space and to minimize the coverage of a self space. If the median distance to the detectors k-nearest neighbors is less then r, a match is detected and matching detectors are discarded. Surviving detectors are then sent to a multi-layer classifier. They authors were able to conclude that scaling is not a problem in NS when real values are used. [Gonzalez et al. 2005]'work hybridized the RNS algorithm with a Self-Organizing Map (SOM). This work attempted to visualize anomalies in 2-dimensional map. In contrast, [Ji and Dasgupta 2004] further extended the RNS algorithm by introducing the variable lengths of a detector radius. They aimed to show an improvement in the detection accuracy and algorithm efficiency, through covering a non-self space with fewer detectors, and cover the holes by using detectors with a smaller radius. One of the problems of using the RNS algorithm is that the number of detectors required to cover a non-self space and the radius of each detector cannot be estimated in advance, and there is no guarantee of achieving the optimal space coverage with minimum overlap. In order to solve these problems, a randomized real-valued negative selection (RRNS) algorithm was introduced by [Gonzalez 2003; Gonzalez, Dasgupta and Nino 2003]. The RRNS algorithm uses Monte Carlo integration, which is a well-known randomized algorithm, to calculate the number of detectors needed to cover a non-self space. It first estimates the volume of a self space based on the assumption that the average minimum distances from collected self samples forms the boundary of self space. Then the number of detectors required to cover a non-self space is calculated by defining a fixed length of detector radius, through obtaining the volume of a non-self space as the complementary to the volume of an estimated self space. Furthermore, simulated annealing is used to minimize the overlapping spaces covered by detectors. The RRNS algorithm was able to provide better non-self space coverage with the same or less computational effort compared to the RNS algorithm.

[Shapiro, Lamont, and Peterson 2005] generated hyper-ellipsoid detectors, which used an evolutionary algorithm that reshaped randomly generated hyper-ellipsoid detectors fit to a non-self space. In contrast, [Ji and Dasgupta 2005] attempted to solve the coverage problem by integrating the statistical hypothesis test to the negative selection algorithm. In this approach, the generation of detectors terminates when the hypothesis test rejects the null hypothesis "The coverage of non-self space by all the existing detectors is below an expected percentage".

Finally, hybrid approaches that combine NS with other algorithms are becoming more common in recent literature. [Dozier et al. 2004; Hou and Dozier 2005] used a steady-state genetic algorithm (GA) to discover the coverage of holes of LYSIS. Their system (GENERTIA) generates additional detectors that can cover the holes discovered by the steady-state GA. [Hang and Dai 2004] [Hang and Dai 2005] used anomaly patterns as seeds to generate additional synthetic anomalies. Artificial anomalies are generated by using a co-evolutionary GA and the NS algorithm. The co-evolutionary GA abstracts the positive selection process of the HIS, which generalizes patterns of the self-class. Then, new artificial anomaly patterns are generated from empty spaces which neighbor a small number of anomaly patterns. The new patterns are then given to the negative selection algorithm with the evolved normal patterns to finalize an artificial anomaly set.

To tackle the scalability problem, an approach based on the linear-time algorithm [D'haeseleer, Forrest and Helman 1996] has been utilized that uses a greedy algorithm that removes redundant detectors, and employing diverse ways of evolving detectors [Ayara et al. 2002; Gonzalez and Cannady 2004]. Another group concentrated on employing a new matching function, namely r-chunks matching [Balthrop, Forrest and Glickman 2002; Esponda, Forrest and Helman 2004], possibly saving computation time during detector generation and matching. Several methods have been investigated to increase the non-self space coverage of detectors. For example, matching [Hofmeyr 1999; Hofmeyr and Forrest 1999a; Balthrop et al. 2002] investigated reducing the number of holes existing in a binary detector space coupled with contiguous matching. [Gonzalez 2003; Ji and Dasgupta 2004] proposed a real-valued detectors with corresponding matching functions. Significant work on a formal framework for positive and negative detection schemes was reported in [Esponda, Forrest and Helman 2004]. This work analyses the trade-offs between two schemes and hence estimates how many self strings are required for either negative or positive detection to insure that it is computationally advantageous. However, the most controversial problem of employing the NS algorithm is based on its initial theory which is self-non-self discrimination and that foreign patterns are detected as intrusions [Aickelin et al. 2003; Burgess 1998]. Non-self patterns would not necessarily indicate intrusions and thus a high false positive error rate caused from this assumption limits the benefits of employing the NS algorithm. Many are trying to tackle this limitation by applying more flexible boundaries between self and non-self space using fuzzy rules such as [Gonzalez 2003; Gomez, Gonzalez and Dasgupta 2003]. However, [Stibor et al 2005] [Stibor, Timmis and Eckert 2005] [Stibor 2006] pointed out that there may be inherent problems with the computational efficiency of NS that can never be resolved.

4.6.3. Danger Theory

The Danger Theory can be beneficial with the artificial immune systems because it does not describe the way the AIS should represent data but which data should be represented. The focus should be on dangerous and interesting data. Danger is usually a grounded signal, and non-self is a set of feature vectors. Therefore, the danger signal helps in identifying which subset of feature vectors is of interest and overcomes many of the limitations of self–non-self selection. With danger theory, the domain of non-self can be restricted to a manageable size, there is no need to screen against all self, and can deal with scenarios where self (or non-self) changes over time. One of the challenges faced when trying to employ the danger theory is to define a suitable danger signal. In general, the human body deals with this issue by responding to the interaction between antigen presenting cells and various signals. The antigen-presenting cells (APCs) activate according to the balance of apoptotic and necrotic cells and this activation leads to protective immune responses. Similarly, the sensors in intrusion detection systems report various low-level alerts and the correlation of these alerts will lead to the construction of an intrusion scenario [Aickelin and Dasgupta 2005].

[Aickelin and Cayzer 2002] explain how an immune response would behave according to the Danger Theory. A cell that is in distress sends out an alarm signal while APC are collecting and capturing antigens that are in the neighborhood. Essentially, the danger signal establishes a danger zone around itself. Thus B cells producing antibodies that match antigens within the danger zone get stimulated and undergo the clonal expansion process. Those that do not match or are too far away do not get stimulated. In general, the danger signal may be a 'positive' signal (for example heat shock protein release) or a 'negative' signal (for example lack of synaptic contact with a dendritic antigen-presenting cell). They also revised Matzinger's [Matzinger 1994] view of danger theory as shown in Figure 4.5. They added a fourth signal (signal 3) which is sent by Th to other APCs in response to detecting bacteria and being under stress. These APCs then are responsible for sending signal 2 to Tk cells. APC not only receives signal 3 from Th cells but also can receive it from viruses.

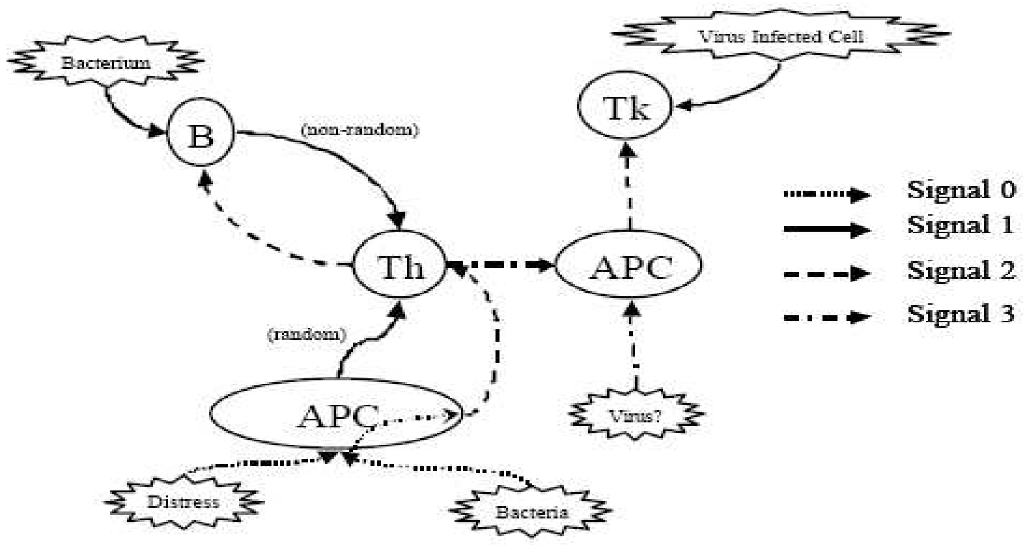


Figure 4.5. Modification of the Danger theory viewed as immune signals [Aickelin and Cayzer 2002]

4.6.3.1. Antigen Presenting Cells (APCs)

It is believed that danger signals are detected and processed through 'professional' antigen presenting cells known as Dendritic cells. Dendritic cells are viewed as one of the major control mechanisms of the immune system, influencing T-cell responses, and acting as an interface between the innate and adaptive immune systems. The Danger Theory rests on the detection of endogenous signals. Endogenous danger signals arise as a result of damage or stress to the tissue cells.

According to the Danger Theory, the pathogens detected are the ones that induce necrosis and cause actual damage to the host tissue. It is proposed that the exposure of antigen presenting cells to danger signals modulates the cells' behavior, ultimately leading to the activation of naive T-cells in the lymph nodes. Alternatively, the absence of danger signals and the presence of cytokines released as a result of apoptosis can lead to antigen presentation in a different context, deleting a matching T-cell. DCs have the capability to combine signals from both endogenous and exogenous sources, and respond appropriately.

Different combinations of input signals can ultimately lead to the differentiation and activation of T-cells. DCs exist in a number of different states of maturity, dependent on the type of environmental signals present in the surrounding fluid. They can exist in immature, semi-mature or mature forms. Immature DCs reside in the tissue where they collect antigenic material and are exposed to exogenous and endogenous signals.

Based on the combinations of signals, mature or semi-mature DCs are generated as in Figure 4.6. Mature DCs have an activating effect while semi-mature DCs have a suppressive effect [Greensmith, Aickelin and Cayzer 2005].

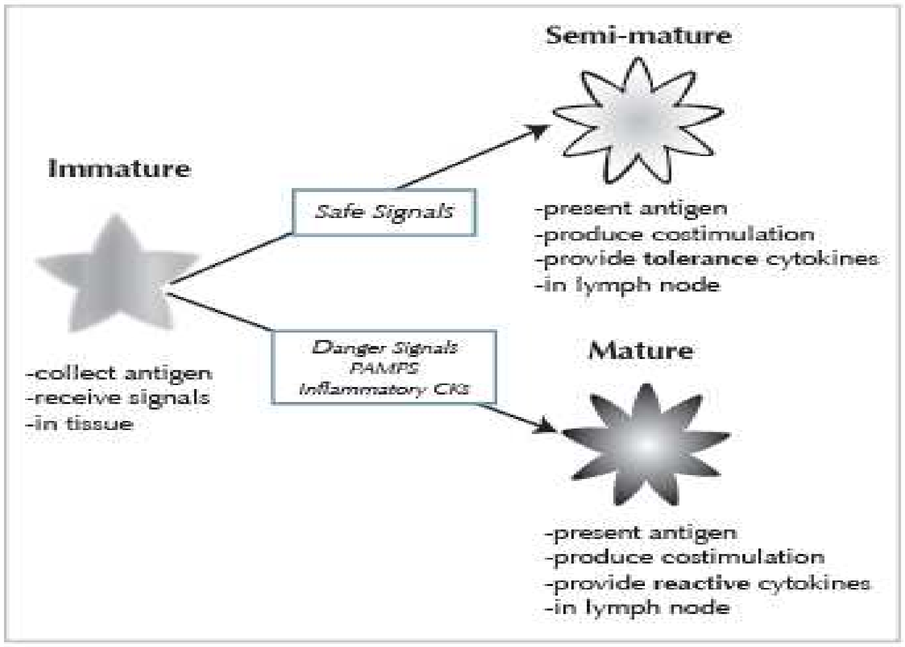


Figure 4.6. The iDC, smDC and mDC behaviors and signals required for differentiation. CKs: denote cytokines. [Greensmith, Aickelin and Cayzer 2005]

In Greensmith, Aickelin and Cayzer's [Greensmith, Aickelin and Cayzer 2005] system, DCs are treated as processors of both exogenous and endogenous signal processors. Input signals are categorized as PAMPs (P), Safe Signals (S), Danger Signals (D) or Inflammatory Cytokines (IC) and represent a concentration of signal. They are transformed to output concentrations of costimulatory molecules (csm), smDC cytokines (semi) and mDC (mat) cytokines. The signal processing is used with the empirically derived weightings. These weightings represent the ratio of activated DCs in the presence and absence of the various stimuli e.g. approximately double the number of DCs mature on contact with PAMPs as opposed to Danger Signals. Additionally, Safe Signals may reduce the action of PAMPS by the same order of magnitude. Inflammatory cytokines are not sufficient to initiate maturation or presentation but can have an amplifying effect on the other signals present. This function is used to combine each of the input signals to derive values for each of the three output concentrations, where C_x is the input concentration and W_x is the weight.

In general, a DC can only collect a finite amount of antigens; therefore, an antigen collection threshold must be incorporated so a DC stops collecting antigen and migrates from the sampling pool to a virtual lymph node. On migration to the virtual lymph node, the antigens contained within an individual DC are presented with the DC's maturation status. If the concentration of mature cytokines is greater than the semi-mature cytokines, the antigen is presented in a 'mature' context. It is possible to count how many times an antigen had been presented in either context to determine if the antigen is classified as anomalous [Greensmith, Aickelin and Cayzer 2005].

The four signals PAMP, danger signals, safe signals and IC [Greensmith, Twycoross and Aickelin 2006][[Greensmith and Aickelin 2006] can be incorporated into a model, implementing DC, each from a different source and producing different output cytokines as follows:

- PAMPS (P) are based on pre-defined signatures. Exposure to PAMPS causes an increase in mDC cytokines. PAMPs are suppressed by safe signals. They cause the maturation of immature DCs to mature DCs through expression of 'mature cytokines'.
- Danger signals (D) cause an increase in mDC cytokines. Danger Signals can also be suppressed by safe signals. Danger signals have a lower potency than PAMPs. Danger signals are released as a result of damage to tissue cells, also increasing mature DC cytokines, and having a lower potency than PAMPs.

- Safe signals (S) cause an increase in smDC cytokines and have a suppressive effect on both PAMPS and danger signals. Safe signals are released as a result of regulated cell death and cause an increase in semi-mature DC cytokines, and reduce the output of mature DC cytokines.
- Inflammatory cytokines (IC) amplify the effects of the other three signals, but are not sufficient to cause any effect on DCs when used in isolation.

The DCA [Greensmith, Aickelin and Twycross 2006] is a population based algorithm, with a user defined number of DCs created to form a sampling pool. While in the sampling pool, each DC is exposed to current signal values and selects a slot in the antigen store. If an antigen is present in the antigen store, the DC collects the antigen and ingests it in the DC internal antigen storage. Each DC has the opportunity to sample multiple antigens. For every iteration of antigen collection, each DC re-calculates its internal cytokine values based on the input signals received. Each antigen can be sampled single or multiple times. Migration is simulated by the removal of a DC from the pool. At this point, the output cytokines of each DC are measured. Antigen presented by cells expressing mature cytokines is labeled 'mature context antigen'. Antigen from cells expressing semi-mature cytokines is labeled as 'semi-mature'. Each presented antigen's context is recorded and eventually a mean antigen context value (between 0 and 1) is derived.

4.6.3.2. Innate and Adaptive immunity

In innate immunity [Twycross and Aickelin 2005; Medzhtov and Janeway 2002], cells are the principal actors in the immune system. Many immune system cells have access to their environment on two levels: the level of antigen and the level of signals.

Antigens are used by the immune system to sense the structure of its environment. The structure is tightly coupled to the context of the environment, which is reflected by levels of signals. Signals reflect what entities are doing on a structural level. There exist many differences between the innate and adaptive immune systems. The adaptive immune system is organized around two classes of cells: T cells and B cells, while the cells of the innate immune system are much more numerous, including natural killer (NK) cells, Dendritic cells (DCs), and macrophages. The environment of a cell is the tissue in which it is located. Tissue is formed by specialized groups of differentiated cells, and forms major components of organs. Cytokines are secreted molecules which mediate and regulate cell behavior, two important subsets of which are tissue factors, inflammation-associated molecules expressed by tissue cells in response to pathogen invasion, and chemokines, cytokines which stimulate cell movement and activation.

Twycross and Aickelin's [Twycross and Aickelin 2005] artificial system is based on populations of interacting agents where cells are seen as autonomous agents. An artificial tissue, in which these agents exist provide an environment in which agents can interact via signaling. As well as passing signals between agents, mechanisms such as antigen processing and presentation to Th cells by DCs suggest the need for agents with the ability to "consume" process and pass on information to other agents. The tissue according to the authors should also provide the services of presenting pathogens at multiple levels. In general, the innate immune system relies on sensing the behavior as well as structure of pathogens.

Accordingly, to adopt danger signals (apoptosis and necrosis) which trigger artificial immune responses within an AIS, [Bentley, Greensmith and Ujin

2005] introduced the concept of artificial tissue. The authors stressed that the tissue is an integral part of immune function, with danger signals being released when tissue cells die under stressful conditions. They also highlighted that the tissue could play the role of interface between immune responses and pathogenic attacks. The authors argued that the absence of artificial tissue in conventional AIS caused difficulties, with every new AIS needing to be “wired” to a specific problem. This makes it difficult to compare, analyze, and apply such existing AIS to new problems. The authors proposed new tissue growing algorithms designed for AIS that provided generic data representations and hence allowed the artificial tissue to play the role of an interface between a problem and an immune algorithm. The algorithms took a series of input data stream formulating the tissue into a specific shape by linking input data cells. When new input data was provided to the tissue, the structure of the tissue changed in response. If danger signals are generated in a tissue, the tissue would provide a spatial and temporal structure, enabling the AIS to start immune responses which were spatially and temporarily focused.

Libtissue [Twycross and Aickelin 2006b] is a software system which allows researches to implement and analyze novel AIS algorithms and apply them to real-world problems and has a client/server architecture. An AIS algorithm is implemented as part of a Libtissue server, and Libtissue clients provide input data to the algorithm and response mechanisms which change the state of the monitored system. This client/server architecture separates data collection by the Libtissue clients from data processing by the Libtissue servers and allows for relatively easy extensibility and testing of algorithms on new data sources. Libtissue is implemented as a library which allows algorithms to be compiled and run on other machines with no modification. Client/server communication is socket-based. AIS algorithms are implemented within a Libtissue server as multi-agent systems of cells. Cells exist within an environment, called a tissue compartment, along with other cells, antigen and signals. The problem to which the algorithm is being applied is represented by Libtissue as antigen and signals. Libtissue allows data on implemented algorithms to be collected and logged. Libtissue clients are of three types: antigen, signal and response. Antigen clients collect and transform data into antigen which are forwarded to a Libtissue server. Currently, a systrace antigen client has been implemented which collects process system calls (syscalls). Signal clients monitor system behavior and provide an AIS running on the tissue server with input signals. A process monitor signal client that monitors a process and its children and records statistics such as CPU and memory usage, and a network signal client that monitors network interface statistics such as bytes per second, have been implemented.

Twycross and Aickelin [Twycross and Aickelin 2006a] also implemented an algorithm to validate Libtissue and have two types of cells, labeled type 1 and 2. Type 1 cells are designed to emulate two key characteristics of biological APC cells: antigen and signal processing. In order to process the antigen, each type 1 cell is equipped with a number of antigen receptors and producers. A cytokine receptor allows type 1 cells to respond to the value of an external signal. Type 2 cells emulate three of the characteristics of biological T cells: cellular binding, antigen matching, and response to antigen. To accomplish this, each type 2 cell has a number of cell receptors specific for type 1 cells, receptors to match antigen, and a response producer which is triggered when antigen is matched. A tissue compartment is created and populated with a number of type 1 and 2 cells. The tissue compartment also stores antigen and signals received from Libtissue clients, which provides the input data to the system. Type 1 cells ingest antigen through their antigen receptors and present it on their antigen producers. The period for which the antigen is presented is determined by a signal read by a cytokine receptor on these cells. Type 2 cells attempt to bind with type 1 cells via their cell receptors. If bound, receptors on these cells interact with antigen producers on the bound type 1 cell. If an exact match between a receptor lock and antigen producer key occurs, the response producer on type 2 cells produces a response.

Previously, the innate immunity has been modeled in a layered architecture as the first layer of defense, and the adaptive as the second layer. However, Twycorss [Twycross 2007], models the innate immune system as the controller of the adaptive immune system. Singh and Nair [Singh and Nair 2005] outline a robot controller based on a combination of the innate and adaptive immune systems. They test their approach on a robotics problem in which a learner robot must learn to accurately follow a track. It can sense when it is on the track and when it looses it. If it looses the track, it first tries to find it on its own and then requests the assistance of a helper robot, who will guide it back to the track. The general idea is to have the learner robot learn to navigate weak portions of the track autonomously, without losing the track and having to be guided back by the helper. The proposed immune system has two type of response governed by separate innate and adaptive subsystems. As the learner travels around the track it sees the track through a simple onboard infrared sensor, and is able to determine when it is on the track, losing the track, or has lost the track. The adaptive component uses a clonal selection algorithm to determine the optimal velocity when the learner senses it is losing the track. The innate immune system, which uses a behavior arbitration mechanism, is activated when the learner senses it has lost the track.

The system of [Kim et al. 2005a] captures syscalls (antigens) by using a system call policy checker tool. The cooperative automated work response and detection immune algorithm (CARDINAL) [Kim et al. 2005b] system consists of periphery and lymph node processes. Both processes reside on a monitoring host and any host running these two processes becomes a part of an artificial body which CARDINAL monitors. The periphery is comprised of DCs and various types of artificial T cells and they directly interact with input data that exists as a part of the periphery. DCs gather and analyze the input data and carry their analysis results to the lymph node. At the lymph node, naïve T cells are created which subsequently differentiate into various types of effectors T cells based on the input data analysis results continuously passed from DCs. Within CARDINAL, effector T cells are automated responders that react to worm related processes in the periphery. Effector T cells are assigned to a response target, a response type, and the number of peer hosts polled. Before the effector T cells migrate from the lymph node to the periphery, they interact with other effector T cells passed from peer hosts. This interaction allows locally generated effector T cells to determine whether they should perform assigned types of responses or not, and the numbers of peer hosts to be polled if they decide a response is appropriate.

The work of Burgess [Burgess 1998] is inspired by the Danger Model. Dangerous programs are detected by the damaging effects they have on the system. Burgess makes the analogy between program termination and biological apoptotic or necrotic cell death. Programs that terminate normally usually generate a SIGCHILD signal, whereas programs that terminate abnormally often generate a SIGABRT or SIGSEGV signal. Normal or abnormal process termination signals can be seen as similar to signals produced by biological cells undergoing apoptosis or necrosis respectively. Burgess has developed Cfengine [Burgess 2000], an autonomous agent and a middle-to-high level policy language for building expert systems to administrate and configure large computer networks. In Burgess's adapted danger model, emphasis of AIS are put on an autonomous and distributed feedback and healing mechanism, triggered when a small amount of damage could be detected at an initial attacking stage. Cfengine automatically configures large numbers of systems on a heterogeneous network with an arbitrary degree of variety in the configuration. After a human administrator initially specifies configuration policies at a very general level using an expert system shell, the system automatically monitors the state of each system and adapts specified policies. Any change in a policy immediately triggers the modification of other policies affecting different hosts. An agent framework which employs an expert system that locally optimizes the maintenance of each local host in a distributed environment is used. [Burgess 2000; Burgess 2001] reports that using Cfengine would save administrator's time, scales well and imposes minimum load. When Cfengine runs, it applies a configuration policy suitable for the classes of monitoring hosts and resources. The class based generic policy is then locally optimized as Cfengine continues to change the policy depending on what is locally observed.

A sophisticated anomaly detection engine was added recently to Cfengine along with several new features [Burgess 2002; Burgess 2004a; Burgess 2004b]. A statistical filter using a time-series prediction was used to detect the significance of deviation. The symbolic content of observed events determines how the system should respond. A statistical anomaly was considered a danger signal and the content of the observed events characterizes the internal degree of the signal. Scalability of the anomaly detection component is increased by incrementally updating the mean and variance of the sampled events. Usually events may represent the number of users, the number of processes, average utilization of the system (load average), and number of incoming and outgoing connections based on each service. Furthermore, [Begnum and Burgess 2003] extended Cfengine by employing the mechanism from pH [Somayaji 2002]. They combined signals from the two systems and intended that pH would be able to adjust its monitoring level based on inputs from Cfengine, and Cfengine would be able to adjust its behavior in response to signals from pH.

[Sarafijanovic and Le Boudec 2004] extended their earlier work on mobile ad-hoc networks [Le Boudec and Sarafijanovic 2003; Le Boudec and Sarafijanovic 2004; Sarafijanovic and Le Boudec 2003] and considered packet loss in the network as a danger signal. In their system the danger signal is used to stop the relevant antigens entering the NS process. The sequences collected at the nodes belonging to the route at the time and where the packet loss is observed are considered as non-self antigens. These non-self antigens are not passed to the detector generation process of the NS algorithm. In addition, danger signals are used as co-stimulation signals confirming successful detection through a detector. Good performing detectors become memory detectors. Their results indicated that the use of danger signals strongly impacted on the reduction of false positive error rates and that adding memory-detectors also improved detection rates.

Pagnoni and Visconti's [Pagnoni and Visconti 2005] NAIS intrusion detection system is inspired by innate immune mechanisms. Their immune system is as a multilayer defense system, and the innate immune system as the first line of defense which is able to recognize self quickly. Their system compiles a list of all observed process names during a training period containing only normal usage. A set of 'digital macrophages' is then created which monitor the system and are activated and generate an alert when they observe any previously unseen process name.

4.6.4. Other Algorithms

Although negative selection and the danger theory are the most popular approaches in AIS for intrusion detection, some researchers choose to create AIS based on alternative ideas. [Forrest et al. 1996] aimed to build an IDS based on an explicit notion of self within a computer system. The system was host-based, examining specifically privileged processes. The system collected self-information in the form of root user sendmail (a popular UNIX mail transport agent) command sequences to construct a database of normal commands. Then, sendmail commands were examined and compared with entries in this database. The time complexity for this operation was O (N) where N is the length of the sequence. A command-matching algorithm was implemented and compared with the defined behavior in the database. Intrusions were detected when the level of mismatched exceeded a predefined threshold value.

[Hofmeyr, Forrest and Somayaji 1998] worked on improving anomaly-based IDS. Misbehavior in privileged processes was examined and system call traces were presented in a window of system calls, a value of six selected by a trial-and-error. This window was compared against a database of normal behavior, stored as a tree structure, and then compiled during a training period. If a deviation from normal behavior was seen, then a mismatch was generated. A sufficiently high level of mismatches generated an alert. The system was able to detect all intrusions, scaled well as well as being able to the find the optimum sequence length and mismatch threshold. The results suggested that this approach could work using data from both real and controlled environments.

[Stillerman, Marceau and Stillman 1999] introduced an immunity-based intrusion detection approach that was particularly applicable to Common Object Request Broker Architecture (CORBA) applications. CORBA is a popular common messaging middle-ware that enables the communication of distributed objects for distributed applications. The authors employed the same approach reported in [Hofmeyr, Forrest and Somayaji 1998] to detect a misuse attacks performed by a legal user of the system. The experimental results showed that the system was able to detect anomalies caused by this attack without high false positive error rates.

[Dasgupta 1999] provided the conceptual view and a general framework of a multi-agent anomaly based intrusion detection system and response in networked computers. The immunity based agents in the system roamed around nodes and monitored network situation. Each agent can recognize others activities and can take appropriate actions according to its predefined security policies. The agent can adapt to its environment dynamically and can detect novel and known attacks. Network activities were monitored on the user, system, process and packet levels.

4.7. AIS Based Intrusion Detection Systems – Summary

There are several systems implemented utilizing one or more immune-inspired algorithms or concepts. Table 4.1. summarizes the relationship between artificial immune algorithms employed to implement different AIS systems and the corresponding biological immune features that inspired such development [Kim et al. 2007].

Table 4.2. presents the AIS based IDSs coupled with the artificial immune algorithms and concepts that were used.

In general, the most commonly used mean of implementing an immune system is through the use of a self-non-self model.

Human Immune Features	**Artificial Immune Algorithms/Concepts**
Distributed	Idiotypic Immune Network, Multi-Agent Systems, Negative Selection
Multi-layered	Multi-Agent Systems, Co-Stimulation
Self-Organized	Gene Library Evolution, Clonal Selection, Negative Selection, Local Sensitivity by Cytokine
Lightweight	Memory Cells, Imperfect Detection, Dynamic Cell Turnover
Diverse	MHC(Permutation Mask)
Disposable	Cell Life Span,
Self/Non-Self Detection	Negative Selection, Tolerization Period

Table 4.1. The Relationship between Biological Immune Features and Artificial Immune Algorithms [Kim et al. 2007]

AIS	Multi agent	Negative selection	Co-stimulation	Gene libraries	Clonal selection	Local sensitivity	generalized detection	Dynamic cell turnover	Permutation mask	Cell life span	Tolerization period	Immune memory	Idiotypic immune networks	Response	Self-non-self
[Forrest et al. 1996]															X
[Hofmeyr, Forrest and Somayaji 1998]															X
[Hofmeyr 1999]		X	X			X	X	X	X	X	X	X			X
[Balthrop, Forrest and Glickman 2002; Balthrop et al. 2002]		X	X				X	X	X	X	X				X
[Kephart 1994; Kephart et al. 1998]	X						X							X	
[Burgess 2000; Burgess 2001; Begnum and Burgess 2003]	X		X											X	X

Table 4.2. Summary of immune-based algorithms used by the complete systems [Kim et al. 2007]

CHAPTER 5

INVESTIGATING INTRUSION DETECTION SYSTEMS THAT USES TRAILS OF SYSTEM CALLS

5.1. Introduction

Intrusion detection using trails of system calls has been studied extensively over the years. Several immune-inspired learning based approaches for host-based intrusion detection have been developed especially for fixed length subsequences or patterns [Forrest et al. 1994] [Forrest et al. 1996] [Forrest, Hofmeyr and Somayaji 1997] [Hofmeyr and Forrest 2000] [Hofmeyr, Forrest and Somayaji 1998] [Warrender, Forrest and Pearlmutter 1999]. For normal patterns, behavior can be generated by executing an application under various normal scenarios. In such approaches anomalous behavior is detected by analyzing sequences of system calls against normal behavior.

System-calls representing normal behavior are grouped into sequences and a database is constructed and patterns are stored, for example, as a tree during training. During detection phase, sequences of system calls are compared to this database using a Hamming distance metric, and a sufficient number of mismatches generate an alert. No user definable parameters are necessary, and the mismatch threshold is automatically derived from the training data. [Somayaji 2000] developed the immune-inspired process homeostasis (pH) intrusion prevention system which detects and actively responds to changes in program behavior in real-time. In his method, sequences of system calls are gathered for all processes running on a host and compared to a normal database using a similar immune-inspired model. However, if an anomaly is detected, execution of the process that produced the system calls will be delayed for a period of time. Similar to the process of generating fixed length detectors, variable length patterns [Jiang, Hua and Oh 2003] [Jiang, Hua and Sheu 2002] [Wespi, Dacier and Debar 1999] [Wespi, Dacier, Debar 2000] can be generated to represent normal behavior.

Many proposals for host-based anomaly intrusion detection can be found in literature. There are those that are based on system call sequences [Forrest et al. 1994][Forrest et al. 1996][Hofmeyr, Forrest and Somayaji 1998][Somayaji 2002][Somayaji and Forrest 2000][Warrender, Forrest, and Pearlmutter 1999][Wespi, Dacier and Debar 2000], data mining [Lee and Stolfo 1998][Lee, Stolfo, and Mok 1999], neural networks [Ghosh, Schwartzbard and Schatz 1999], finite automata [Michael and Ghosh 2000], hidden Markov models [Ourston 2002], and pattern matching in behavioral sequences [Lane and Brodley 1997][Lane and Brodley 1999].

5.2. Experiment Setup

Three systems that are based on system call sequences are chosen and are implemented with Microsoft Visual Studio 2005 as a win32 console application. Sequence method based IDS, lookahead-pairs method based IDS and variable length with overlap relationship method based IDS have been implemented. For off-line testing of the implemented algorithms in this dissertation, the login application was investigated and its input data sets for both training and testing were obtained from the website of the

University of New Mexico (http://www.cs.unm.edu/~immsec). The login data set was used to detect Trojan horse attacks.

At this web site there are several trace files. Each trace is the list of system calls issued by a single process from the beginning of its execution to the end. Trace lengths vary widely because of differences in program complexity and because some traces are daemon processes and others are not. Each trace file (*.int) lists pairs of numbers, one pair per line. The first number in a pair is the PID of the executing process, and the second is a number representing the system call. Note that there may be multiple processes within a single file, and they may be interleaved.

Data input pre-processing was conducted on these files to make them suitable for processing by the three systems. To ensure unity across results, the same files were inputted to the three systems in the same order and in the same format. There were two normal execution logs of the login application. Each log consists of the traces of multiple processes interleaved in the same log file. In general, each line in the log file listed pairs of numbers. The first number is the PID of the executing process and the second number is the ID of the system call. We performed pre-processing to the two log files to group entries of the same PID together in one file in the same order they appeared in the original log file. As a result, the first log contained 16 traces and the second contained 8 traces. For example, the log files resulted in several files such as int_509.txt, int_531.txt, int_625.txt, etc. where “int” indicates that the file hold integer values. The number, such as “531”, is the PID and “.txt” is the file type. Furthermore, the file “int_531.txt” holds only integer values of the system calls that have been carried out by the process with the PID= 531. All traces were used to train the 3 implemented systems and to generate their pattern databases. To test the systems, a stricter test of Trojan horse attack that was designed by the University of New Mexico called “home-grown” was used. During evaluation, the testing input file is read one system call at a time. It goes through a pre-processing stage which translates a system call to its corresponding ID and checks its availability in the pattern DB.

5.3. Sequence Profile Method

5.3.1. Background Information

According to [Forrest et al. 1996] [Hofmeyr, Forrest and Somayaji 1998], a set of system call sequences that can be produced by an application can be specified. These sequences are determined by the ordering of system calls in the set of the possible execution paths through the program text during normal execution. Despite being huge, the short range ordering of system calls appears to be consistent, and can be used to define normal behavior. To build up the normal database, a window of size k is slid across the trace of system calls recording which system call follow which within the sliding window. For example, the sequence of system calls {open, read, mmap, mmap, open, getrlimit, mmap, close} with a window size k=4 will produce a database shown in Table 5.1. In general, the input sequence of system calls will be scanned, one system call at a time, storing the current system call and a number of system calls up to window size k following this current system call. Each sub-sequence is stored as a row in a temporary table until all sequences have been processed. Entries (rows) in the table may appear more than once, such repeated entries are removed, keeping only one instance of the sub-sequence. Furthermore, entries starting with the same system call are grouped together as shown in Table 5.2. Finally, entries in the final table are stored in a tree structure. Each current system call is the root of the tree and the children of the root are expended depending on which system call appears next while training. .

In the testing phase, when comparing against the normal system call profile, the sequence {open, fstat, mmap, execve} will be signaled as anomalous because this sequence is not listed in the normal database. There are many ways to reject this sequence, depending on the security requirements of the application that is monitored. Usually a mismatch threshold is associated with anomaly identification. For example, with a window size k = 4 and the sequence {open, fstat, mmap, execve}, four threshold values could be employed depending on how many mismatch system calls within the sequence should be anomalous to flag an intrusion. If we have high security requirements, only one anomalous system call within the sequence will flag an intrusion. However, if we are more lenient, then 2 or even 3 system calls can be flagged as a mismatch and still not be considered as an intrusion.

Current system call	**Position 1**	**Position 2**	**Position 3**
open	read	mmap	mmap
Read	Mmap	Mmap	Open
Mmap	Mmap	Open	Getrlimit
Mmap	Open	Getrlimit	Mmap
Open	Getlmimit	Mmap	close
getrlimit	mmap	close	
mmap	close		
close			

Table 5.1. Expanded database produced when K= 4 and for the normal sequence {open, read, mmap, mmap, open, getrlimit, mmap, close} [Forrest et al. 1996][Hofmeyr, Forrest and Somayaji 1998].

Current system call	Position 1	Position 2	Position 3
Open	read	mmap	mmap
	getrlimit	mmap	close
Read	mmap	mmap	open
Mmap	mmap	open	getrlimit
	close		
	open	getrlimit	mmap
Getrlimit	mmap	close	
Close			

Table 5.2. Grouping entries that start with the same system call [Forrest et al. 1996][Hofmeyr, Forrest and Somayaji 1998].

5.3.2 Implementation

The pattern generation idea was adapted from [Forrest et al. 1996]. Forrest et al. stored their pattern normal database as a tree; however our normal patterns- database is implemented as a hash table of the size of NUMBER_SYSCALLS representing the total number of system calls. This number can be increased and decreased as desired with no effect on our implementation. Each system call is mapped to an entry in the hash table. Each entry in the table is a pointer to a linked list of all patterns starting with this system call. All sequences are of the same length which is equal to window size.

The datasets used for training the system were obtained from the University of New Mexico. However, our system has the ability to perform a pre-processing stage of translating the system call sequences to their corresponding PID. This is achieved by reading a system call in its original form and translating it according to a hash table to its corresponding PID. The content of the translation hash table is also available at the university's website.

The steps performed in the training phase are as follows:

1. Input training files were read one at a time and either processed immediately or stored in a linked list of linked lists to allow easier processing. The first linked list points to the first system call of each training file. Then the content of each file is stored in another linked list. Storing input in such a data structure is not necessary but facilitates the processing of the input data.
2. A 2 dimensional array is created to store the subsequences generated when applying the sequence method concept depending on a pre-specified window size. The number of columns of this 2D array is equal to the window size and the number of rows is proportional to the number of system calls in all training files and window size.
3. After filling the initial database with the input training file, the entries in the table are scanned to remove redundant entries.
4. Finally, the content of the table are stored in a hash table data structure similar to Figure 5.1. The size of the hash table is equal to the number of system calls that can be generated by an application. Each entry in the table is a pointer to a linked list of a data structure (a 1 dimensional array whose size is equal to the window size) and a pointer to the next pattern (sub-sequence). Entries are added to the hash table as appropriate.

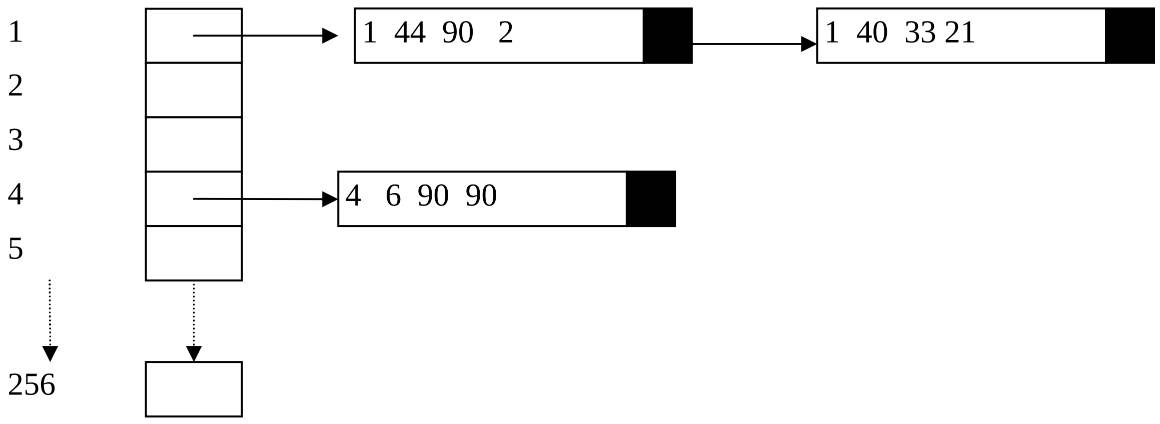


Figure 5.1. Hash table holding the sequence method profile entries. All entries are of equal size and are equal to the window size.

The steps performed in the testing phase are as follows:

1. Start logging testing activities such as time, the subsequence currently under consideration, detection rate, etc.
2. Open the log file containing intrusive signatures for reading.
3. We read one system call at a time until a subsequence of appropriate length (equal to window size) is reached.
4. Compare the current subsequence with entries in the normal database and if it does not appear in the normal database, we increment a counter representing the threshold of maximum read mismatches. The system can display mismatches as they are discovered, or display a mismatch if the total number of mismatches reaches to a threshold.
5. Finally, when we finish processing the tested log file we display the following:
   - Testing duration.
   - Number of sequences handled while testing.
   - Number of sequence anomalies.

  - Mismatch anomaly which indicates the percentage of mismatches with regard to the total number of sequences handled while testing the system.
  - Number of sequences in the normal database.
  - Space cost while running the normal database.
  - Space cost while saved to disk of the normal database.

The following were considered when implementing the system:

- In our experiments, we have scanned the entire log testing file and counted how many times intrusive instances (mismatches) had occurred. Afterwards, we displayed the mismatch anomaly value which is equal to the total number of mismatches found divided by the total number of sequences handled while testing.
- In the hash table of the normal database, each entry points to a linked list of all possible normal patterns seen while training the system. This linked list will be searched completely before deciding if there is a mismatch or not. If, for example, the current tested sequence appears to be similar to a sequence in the normal database but disagree with the last or middle system call. A mismatch will not be declared unless it is the last tested sequence in the linked list. This is because even though this sequence results in a mismatch, another sequence matching the tested sequence may exist afterwards.

Sample code of the implemented sequence method IDS can be found in appendix A.

5.3.3. Performance

In general, there is no correlation between the actual number of system calls in the input training file and the number of sequences in the normal database. This is because repeated patterns will be removed from it.

From table 5.3., we noted that the number of sequences increases as the window size increases but does not follow a linear pattern. Originally, the number of patterns in the database decrease by one as the window size increases by one since more system calls builds a pattern and fewer patterns are required. This is evident from the log file generated by our system. A sample of the output of the testing log files can be found in Appendix B. In this log file we display the number of rows (patterns) before removing redundant data and after. As we increase the widow size from 3 to 4, the number of patterns before removing redundant data decrease by 1 and the number of patterns after removing redundant data increase. For the input file int_509.txt, as shown in Table 5.4., and as the window size is increasing from 3 to 5 we find that the number of rows decreases by one. However, since we are building an efficient data base of normal behavior and because a number of similar patterns exist in the same input training file and among other training files, the final number of rows that are added to the database are different. Such number is affected by the number of redundant rows.

From table 5.3., the space cost while saved to disk is less than while running because we are saving integer values to a file whereas while running we need to consider the hash table and structures holding pattern information. The number of sequences in the testing file decreases by one as the window size is increased by one because removing similar entries are not performed here.

The mismatch % is calculated as the number of total number of mismatches divided by the total number of sequences handled while testing. The reason the mismatch % threshold increases as the window size increases is because the system call that causes a mismatch starts to appear in more sequences created from the testing file. For example, suppose we have the following normal sequence {1, 2, 3, 4, 5, and 6} and testing sequence {2, 9, 3, and 4} the output produced when performing sequence method intrusion detection is shown in Table 5.5... As the window size increases, the system call causing the mismatch starts to appear in more rows from the testing file and the number of sequences in the testing log file tends to decrease. Therefore, the mismatch % which is equal to the number of mismatches divided by the number of sequences in the testing log files starts to increase.

Figure 5.2. shows an increase in the space cost of the system in both “while running” and “while saved to disk” as the number of sequences (patterns) stored in the normal database increases. In general, the space cost while running is higher than while saved to disk because while running we are considering the space cost of the hash table, linked lists, and arrays used to store the different data structures of the normal database. However, we are only saving to disk the content of the patterns which is a list of integer values. When the number of sequences is less than 400, the increase in space cost is slower than the increase in space cost as. This is because the size of the array holding each sequence is larger and the number of arrays required to hold the sequences is larger. Meaning that more arrays are needed to store the pattern sequences and larger arrays are needed to hold the longer sequences as shown in table 5.3.

# sys calls / W	# sequences in normal DB	space cost while running (bytes)	space cost while saved to disk (bytes)	# sequences in testing	Mismatch % threshold
2	142	4548	1136	1349	1.18
3	199	12740	2388	1348	2.22
4	235	22564	3760	1347	3.41
5	264	33796	5280	1346	4.46
6	291	46564	6984	1345	5.65
7	318	61060	8904	1344	6.84
8	341	76388	10915	1343	8.64
9	359	91908	12924	1342	10.13
10	375	108004	15000	1341	11.41
11	389	124484	17116	1340	12.46
12	402	141508	19296	1339	13.52
13	413	158596	21476	1338	14.42
14	423	175972	23688	1337	15.26
15	433	193988	25980	1336	16.09
16	441	211684	28224	1335	16.85
17	447	228868	30396	1334	17.61
18	451	245348	32472	1333	18.3
19	455	262084	34580	1332	18.99
20	459	279076	36720	1331	19.61
21	461	295044	38724	1330	20.15
22	463	311140	40744	1329	20.69
23	465	327364	42780	1328	21.23
24	467	343716	44833	1327	21.7
25	469	360196	46900	1326	22.17
26	471	376804	48984	1325	22.64
27	473	393540	51084	1324	23.11
28	475	410404	53200	1323	23.58
29	477	427396	55332	1322	24.05
30	479	444516	57480	1321	24.53
31	481	461764	59644	1320	25
32	483	479140	61824	1319	25.39

Table 5.3. Sequence method performance across several window sizes.

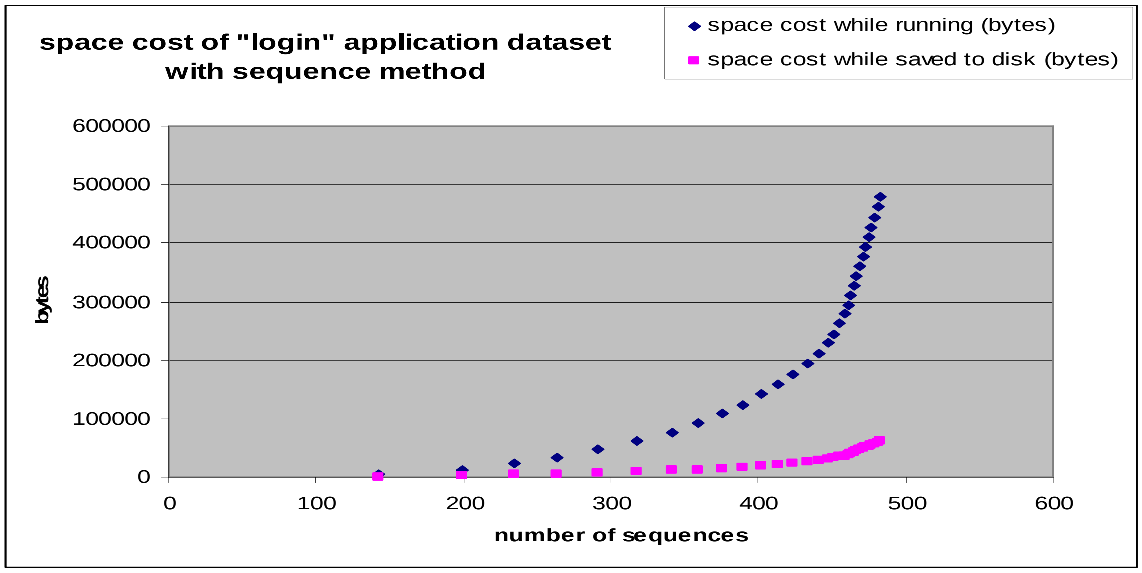


Figure 5.2. Space cost while running and while saved to disk as the number of sequences increases for the "login" application dataset with sequence method.

Window size	# rows before removing redundant data	# rows after removing redundant data
3	362	181
4	361	203
5	360	226

Table 5.4. number of rows before and after removing redundant entries while running sequence method IDS across different window sizes.

W	Rows in normal DB	Rows from testing file	Number of mismatches	Number of sequences in testing log file	Mismatch %
2	1,2 2,3 3,4 4,5 5,6	2,9 9,3 3,4	2	3	66%
3	1,2,3 2,3,4 3,4,5 4,5,6	2,9,3 9,3,4	2	2	100%

Table 5.5. Example showing how the window size affects the value of mismatch % while running sequence method IDS.

5.4. Lookahead-Pairs Profile method

5.4.1. Background Information

Somayaji [Somayaji 2002] attempted to employ a different approach to store the sequences of system calls, called the lookahead-pairs method. With this technique a profile of the program's behavior consists of the pairs formed by the current and a past system call(s) depending on the window size chosen. For example, with a window size w = 4 and the trace of system calls :{ execve, brk, open, fstat, mmap, close, open, mmap, munmap} the generated subsequences are shown in Table 5.6... In Table 5.7., the sequence representation is compressed by joining together lines with the same current value. From this table, three sets of lookahead pairs are generated creating the lookahead-pairs profile database that is shown in Table 5.8... It consists of pairs of the current system call and the system calls in position 1 (placed in row =2) and called set 0, pairs of the current system call and the system calls in position 2 (placed in row =3) and placed in set 1, and pairs of the current system call and the system calls in position 3 (placed in row =4) and placed in set 2.

This table is then stored using a fixed size bit array(s). Each set (row) in the table is stored in a bit array of the size of (NUM SYS CALLS * NUM SYS CALLS). The complete database is stored in multiple array equal to window size and the size of each array is equal to (NUM SYS CALLS * NUM SYS CALLS). To efficiently take advantage of bit manipulation of bytes on Linux and UNIX machines, a window size of 9 or 17 is preferred. This is because a window size of 9 means that we will end up with 8 sets that can be stored in a byte array = 8 bit arrays.

In the detection phase, the sequence {open, fstat, mmap, execve} will be identified as anomalous because the lookahead pairs (execve, mmap) (row = 2), (execve, fstat) (row = 3), and (execve, open) (row = 4) are all not present in table.

position 3	position 2	position 1	current
			execve
		execve	brk
	execve	brk	open
execve	brk	open	fstat
brk	open	fstat	mmap
open	fstat	mmap	close
fstat	mmap	close	open
mmap	close	open	mmap
close	open	mmap	munmap

Table 5.6. a sample profile generated for the system call sequence {execve, brk, open, fstat, mmap, close, open, mmap, munmap} [Somayaji 2002].

position 3	position 2	position 1	current
			execve
		execve	brk
fstat	execve, mmap	brk, close	open
execve	brk	open	fstat
brk, mmap	open, close	fstat, open	mmap
open	fstat	mmap	close
close	open	mmap	munmap

Table 5.7. a sample lookahead pair profile, with the pairs represented implicitly. Note that there are multiple entries in the open and mmap rows [Somayaji 2002].

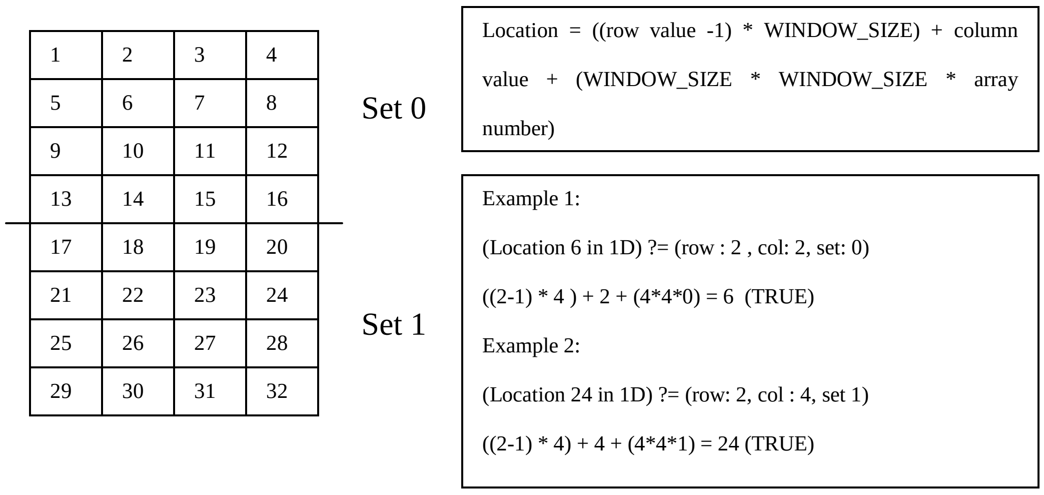
1	pairs (current, prev)
2	(brk, execve), (open, brk), (open, close), (fstat, open), (mmap, fstat), (mmap, open), (close, mmap), (munmap, mmap)
3	(open, execve), (open, mmap), (fstat, brk), (mmap, open), (mmap, close), (close, fstat), (munmap, open)
4	(open, fstat), (fstat, execve), (mmap, brk), (mmap, mmap), (close, open), (munmap, close)

Table 5.8. A sample lookahead pair profile, with the pairs represented explicitly [Somayaji 2002].

5.4.2. Implementation

The patterns' storage idea was adapted from [Somayaji 2002] where all possible lookahead-pairs patterns can be stored in (window size - 1) bit arrays. An ideal window size in Somayaji's implementation was 9 since 9 -1 = 8 sets which can be stored in a c x c byte array, where c is the number of system calls. On the other hand, our implementation took advantage of the bit array in the <bitset> library. Here the values can be stored in a 1 dimensional array format and there is no limitation on the number of sets to store or windows size.

The formula used to access individual points in the array is ((row value -1) * WINDOW_SIZE) + column value + (WINDOW_SIZE * WINDOW_SIZE * array number). Figure 5.3. shows how this formula is used to find the correct corresponding cell in a 1 dimensional array that is equivalent to a value in one of the 2 dimensional bit arrays.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

Location = ((row value -1) * WINDOW_SIZE) + column value + (WINDOW_SIZE * WINDOW_SIZE * array number)

Example 1:

(Location 6 in 1D) ?= (row : 2 , col: 2, set: 0)

((2-1) * 4) + 2 + (4*4*0) = 6 (TRUE)

Example 2:

(Location 24 in 1D) ?= (row: 2, col : 4, set 1)

((2-1) * 4) + 4 + (4*4*1) = 24 (TRUE)

Figure 5.3. Example explaining the mapping equation from one entry in a two dimensional array to a one dimensional array.

If at a given set there is a relation between two system calls, the location is set to ON. At testing time, this location is checked, if in the testing profile two system calls show that there is a relation but their connecting location is set to OFF an intrusion is detected.

In general the following steps are carried out when training the system:

1. Input training files are open for processing.
2. We read one file at a time and we read a sequence whose length is equal to WINDOW SIZE.
3. Ws start creating the pairs by pairing the current system call and the previous system calls. Removing redundant entries is not important because if a repeated entry is processed it will set the cell of the 1 dimensional array again which has no effect.

In the testing phase the following will be performed:

1. The testing file is open for reading. We conduct, on line processing to this file by reading and creating a subsequence equal to WINDOW SIZE, one subsequence at a time.
2. We generate the associated pairs and then test them against the database. If the cell is not set then there is a mismatch.

Sample code of our implementation to lookahead-pairs method intrusion detection system can be found in Appendix C. Sample of a log file of running the lookahead pairs method based IDS can be found in Appendix G.

5.4.3. Performance

Table 5.9. shows the performance of the lookahead-pairs method when the window size is increased from 2 to 32. The number of pairs in the normal DB and testing file and the space cost of maintaining the normal database while running and saved to disk increases as window size increases. In general, the number of pairs of the normal database increases as the window size increases and it depends on how the data in the normal file relate. If more entries are similar then the number of pairs tends to be smaller. Space cost while running is equal to the size of bit array used. Different from the original implementation of lookahead-pairs method, our implementation does not favor a specific window size. The space cost while saved to disk is smaller than while running because in our implementation we are only storing the locations that are set to 1. Of course, if the locations that are set to one increase then the cost while saved to disk will increase having no affect on cost while running. The number of pairs being tested increase as the window size increases because as the sequence length under consideration gets longer the number of pairs generated for it also increases. Finally, the mismatch % is equal to the number of pairs that raise a mismatch divided by the total number of pairs handled while testing.

Figure 5.4. shows an increase in the space cost of the system in both while running and while saved to disk as the number of pairs stored in the normal database increases. The slope of increase for the space cost while running is higher than while being saved to disk. This is because while running, the space cost is equal to the number and size of arrays used to store the relationship between pairs. This includes array locations that are set to 1 (there is a relationship between associated pairs) and set to 0 (there is no relationship between associated pairs). However, the space cost while saved to disk is only saving the locations that are set to 1 which is dependent on how many locations in the arrays are set to one. The increase in number of pairs is also related to the window size as shown in table 5.9.

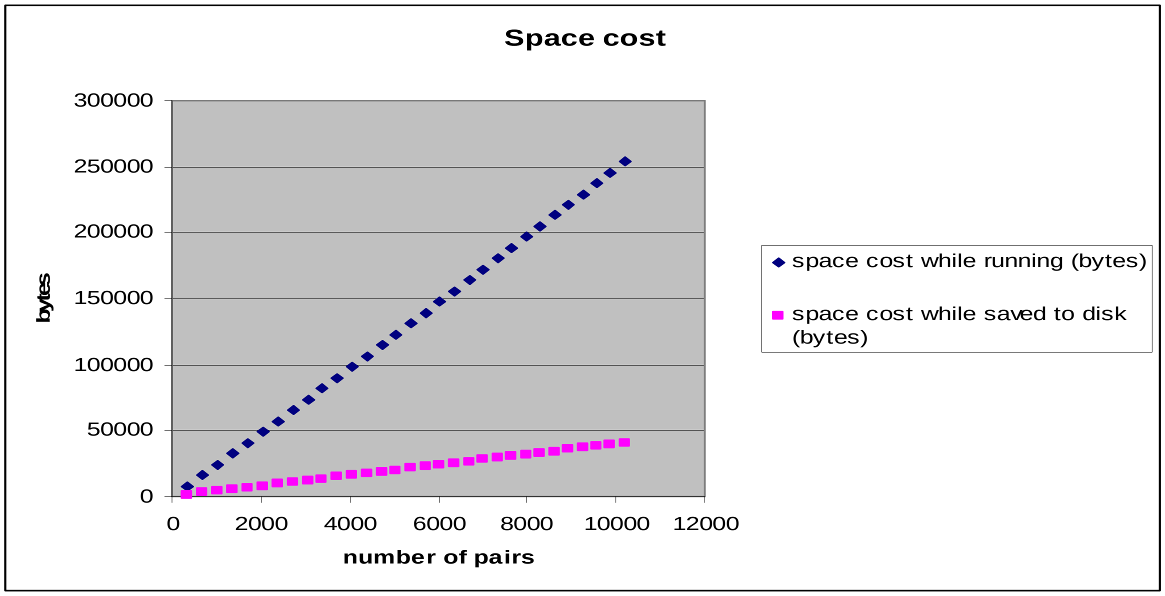


Figure 5.4. Space cost while running and while saved to disk as the number of pairs increases for the "login" application dataset with lookahead-pairs method.

# sys calls / W	# pairs in Normal DB	space cost while running (bytes)	space cost while saved to disk (bytes)	# pairs in testing	Mismatch % threshold
2	344	8192	1376	1349	1.18
3	687	16384	2748	2697	1.44
4	1029	24576	4116	4044	1.66
5	1370	32768	5480	5390	1.95
6	1710	40960	6840	6735	2.29
7	2049	49152	8196	8079	2.65
8	2387	57344	9548	9422	3.01
9	2724	65536	10896	10764	3.39
10	3060	73728	12240	12105	3.77
11	3395	81920	13580	13445	4.06
12	3729	90112	14916	14784	4.34
13	4062	98304	16248	16122	4.56
14	4394	106496	17576	17459	4.78
15	4725	114688	18900	18795	4.91
16	5055	122880	20220	20130	5.01
17	5384	131072	21536	21464	5.12
18	5712	139264	22848	22797	5.26
19	6039	147456	24156	24129	5.37
20	6365	155648	25460	25460	5.49
21	6690	163840	26760	26790	5.56
22	7014	172032	28056	28119	5.71
23	7337	180224	29348	29447	5.84
24	7659	188416	30636	30774	5.97
25	7980	196608	31920	32100	6.11
26	8300	204800	33200	33425	6.26
27	8619	212992	34476	34749	6.4
28	8937	221184	35748	36072	6.56
29	9254	229376	37016	37394	6.65
30	9570	237568	38280	38715	6.78
31	9885	245760	39540	40035	6.89
32	10199	253952	40796	41354	6.99

Table 5.9. lookahead pairs method performance across several window sizes.

5.5. Variable-Length With Overlap- Relationship Profile Method

5.5.1. Background Information

Due to the limitations found with fixed length patterns, Debar et al. [Debar et al. 1998] [Wespi, Dacier and Debar 1999] [Wespi, Dacier and Debar 2000] presented a novel technique to build a table of variable length patterns based on the TEIRESIAS algorithm [Rigoutsos and Floratos 1998]. The system comprises two main parts: an off-line part, which corresponds to training the system and an on-line part, which corresponds to the detection system. In the training phase, system calls are sorted and then translated to an internal format for processing. Consecutive occurrences of the same character are aggregated and duplicate sequences are removed. Finally, the pattern table is generated by the pattern-extraction module.

Jiang et al. [Jiang, Hua and Oh 2003] [Jiang, Hua, and Sheu 2002] proposed an intrusion detection system employing variable-length patterns with overlap relationship. It modifies some limitations in the TEIRESIAS-based method in that: it refines the definition of maximal patterns, identifies overlap relationships between patterns (inter- and intra-pattern anomalies), and does not need a look-ahead threshold. The system consists of two components: offline training and online detection parts. Each component consists of the following modules:

1. Data collection module: for capturing and recording sequences of system calls.

2. Data preprocessing module: to translate system calls to its corresponding ID and perform aggregation on data.

3. Pattern extraction module: to extract maximal patterns.

4. Pattern overlap relationship identification module: organize patterns into adjacency lists and indicate overlap relationship between patterns.

5. Pattern matching module: to identify any deviation.

In pattern extraction, the training sequences are scanned for each never-seen-before system call e. Then the maximal patterns starting with e are identified in an iterative manner. The algorithm first identifies instances of a corresponding system call and assigns each instance an index denoted by a parenthesis as shown in Figure 5.5.(a) for the system call 4. Initially, every instance of each system call e forms a 1-pattern instance. Each i-pattern p is expanded to create i+1-patterns. The instances of each of these patterns are stored in a data structure called pinstance set. Then the system call instances immediately following each occurrence of p in the training set are inspected. Three mutually exclusive types of instances of p can occur:

Type 1: the element of the pinstance set under consideration is at the end of the training sequence and is the last system call in its corresponding sequence.

Type 2: the system call following the current system call in a sequence does not follow the same system call in the other sequences.

Type 3: the system call following this system call also follows at least one other instance of this system call.

Both type-1 and type-2 are considered maximal pattern candidates. Type-1 cannot be further expanded in the forward direction. Type-2 instances can be expanded into i+1-patterns, but they will not be frequent patterns. Each type-3 i-pattern instances can be expanded to create i+1-pattern instances. These i+1-pattern instances are grouped into different pinstance sets according to their last system call.

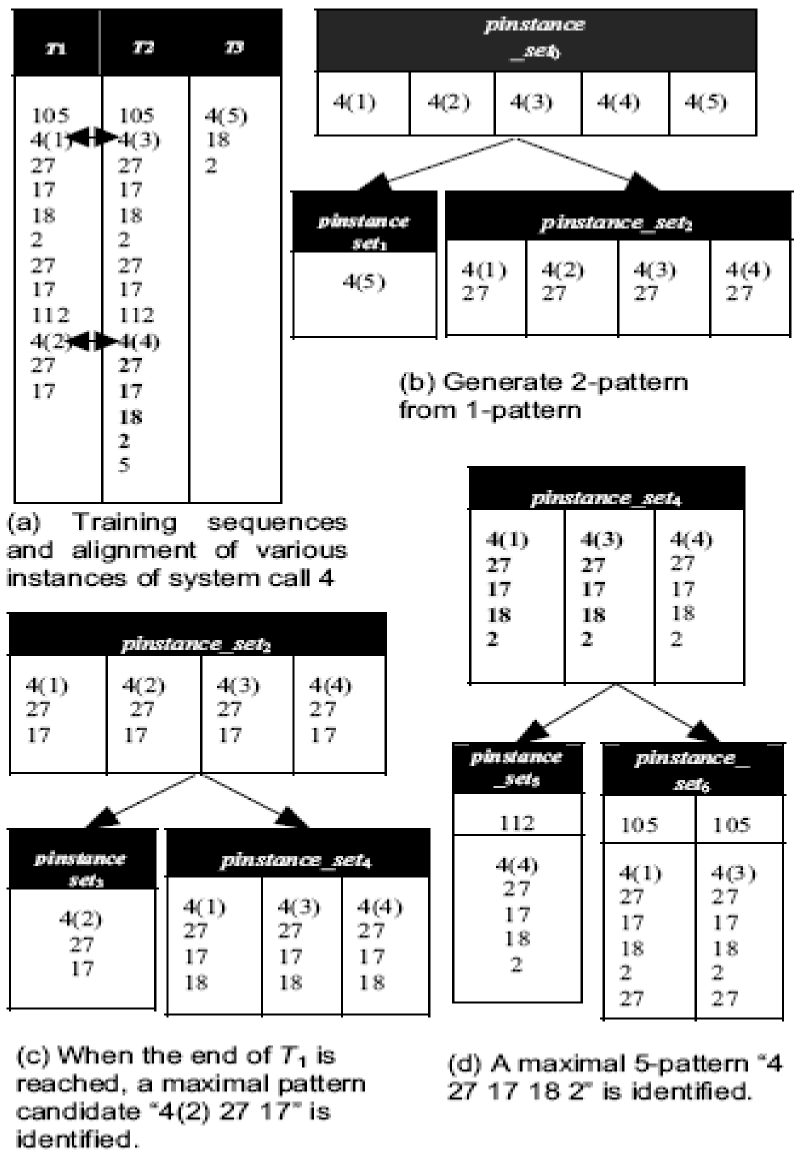


Figure 5.5. Steps of extracting maximal candidate and maximal patterns [Jiang, Hua, and Sheu 2002].

From Figure 5.5(a), there are five instances of system call 4, labeled as 4(1), 4(2), 4(3), 4(4), and 4(5) and they are copied into pinstance_set0 in Figure 5.5(b). From Figure 5.5(a), 4(5) is of type 2 because it is the only one of the five pattern instances followed by system call 18. System call 27 is added to the system calls in pinstance_set2 to expand the remaining pattern instances into 2- pattern instances. The same technique is applied until no more expansion can be performed.

The system call sequence identified in the last pinstance_set6 is considered a maximal pattern. At this point, candidate maximal patterns are examined. To be classified as a maximal pattern, the pattern should 1) not be a subsequence of another maximal pattern and 2) either the system call preceding the system call in the pinstance set does not precede any other instance of p or it is at the beginning of the training sequence. The final stage is to collect standalone instances that participate in 1-patterns by scanning the training sequences and output such patterns in the longest possible form.

5.5.2. Implementation

We adapt the variable-length with overlap-relationship profile generation idea from [Jiang, Hua, and Sheu 2003] [Jiang, Hua, and Sheu 2002]. In general, the process of generating patters is very expensive in terms of processing and memory requirements. However, it is only performed once and the finial generated pattern database can be stored and accessed by the application with no need to repeat this process for the same monitored application. Simply our pattern generation model will perform similar to the following examples.

Example 1:

For the input training files:

T1: {105, 4, 27, 17, 18, 2, 27, 17, 112, 4, 27, 17}

T2: {105, 4, 27, 17, 18, 2, 27, 17, 112, 4, 27, 17, 18, 2, 5}

T3: {4, 18, 2}

We will start scanning from T1 and its first system call 105. We are looking for the longest maximal patterns starting with system call 105. Finally stand alone instances are added to the pattern database.

The following patterns will be added to the database:

P1: {105, 4, 27, 17, 18, 2, 27, 17, 112, 4, 27, 17} appear at least 2 times

P2: {18, 2} appear at least 2 times and although it appears at a subsequence of another pattern, it appears at the end of the training sequence.

P3: {5} stand alone pattern because it appears one time in the training files and does not belong to a pattern.

P4: {4} stand alone pattern because it appears one time in the training files and does not belong to a pattern.

Example 2:

For the input training file:

T1: {90, 7, 2, 3, 90, 6, 1, 4, 90, 7, 2, 3, 90, 6, 1, 4, 90, 3, 5, 2, 6, 90, 3, 5, 1, 90, 3, 5, 2, 6, 90, 90, 115}

The system will divide the training sequence to the different pinstance subsequences with respect to the system call 90.

The following are sample of the subsequences starting with the system call 90:

{90, 7, 2, 3} $\rightarrow$ (1)

{90, 6, 1, 4} $\rightarrow$ (2)

{90, 7, 2, 3} $\rightarrow$ (3)

{90, 6, 1, 4} $\rightarrow$ (4)

{90, 3, 5, 2, 6} $\rightarrow$ (5)

{90, 3, 5, 1} → (6)

{90, 3, 5, 2, 6} → (7)

{90} → (8)

{90, 115} → (9)

Subsequences 1, 2, 3, 4, 5, 7 are maximal sequences and one instance of the repeated patterns is added to the pattern database. Subsequences 6, 8 and 9 are stand alone sequences and are added to the pattern database.

The final database will contain the following patterns:

{90, 7, 2, 3}

{90, 6, 1, 4}

{90, 3, 5, 2, 6}

{90, 3, 5, 1}

{90, 90, 115} the sequences 8 and 9 are combined together since they follow each other in the input training file and they are collected as stand alone instances in their longest form.

Several test cases were used to investigate the performance of the implemented system such as:

- The tested sequence is normal and matches patterns in the database.
- The tested sequence contains a number of mismatches.
- A system call in the tested sequence may not belong to any pattern and is an invalid system call. This system call may appear at the beginning of the tested sequence or in another position. If it is found, the system will continue to the next system call and will start matching against other patterns in the database.

- The tested sequence may expand across several patterns in the database. This can be in different entries in the hash table or in the same entry.
- Contiguous mismatches up to a predefined threshold that are checked against one pattern will be backtracked and checked against another entry.

Our implementation differs from [Jiang, Hua, and Sheu 2002] in that:

1. Since we are doing extensive processing on the input training data, we decided to store its content in a linked list (adjacency lists) of linked lists. Each node in the original linked list holds the contents of one file.

2. Finding maximal patterns required the use of a data structure to hold the different values of pinstance values. We have chosen to implement this process with a tree structure where the left child is a one dimensional array and the right child may hold a number of one dimensional arrays. Left children are not expanded since they represent a maximal pattern candidate. The right children are expanded further until we have a right child holding only two subsequences. The last left child is considered a maximal pattern and all left children are traversed to find another possible maximal pattern. A pattern p is considered maximal if there is no other pattern q that contains p and has the same number of occurrences. Also, if a sequence p appears as a subsequence of q, p can be chosen if its last system call is followed by a NULL value. For example, both {{4, 27, 17, NULL}, {4, 27, 17, 18, 2, NULL}} are considered maximal.

3. Identifying type 2 was not simple as explained in the paper. For example, identifying only one subsequence as type1 or type2 was not true. Furthermore, even if one subsequence was identified as type1 or type2, this did not mean that the

remaining subsequences were of type3. Type3 subsequences created disjoint groups where each group can be expanded and processed separately. We handled, for example, the following training sequences:

{{90 1 2 3 4}, {90 1 2 3 4}, {90 1 2 3 4}, {90 5 6 7 8},

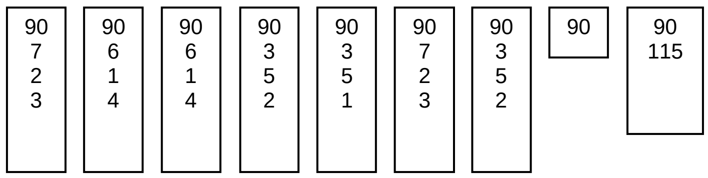
{90 9 10 11 12}, {90 5 6 7 8 9 10 11}, {90 5 6 7 8 9 10 11}}

Such sequences required that the grouped sequences of the subsequence {90 1 2 3 4} be completely processed. Then the system "concurrently" process the grouped sequences of {{90 5 6 7 8}, {90 5 6 7 8 9 10 11}}. This process is repeated until all subgroups- if they exist- are processed.

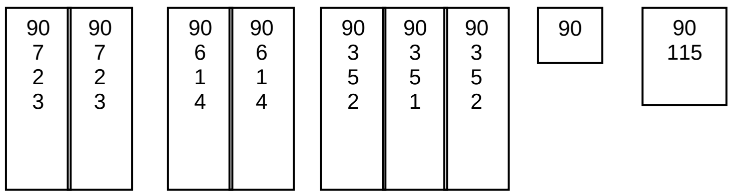
4. The previous example also introduced a situation where not only would sequences be grouped into subgroups but also a subgroup could be further grouped into subgroups. Our solution was to insert each subgroup in a queue and process them until no groups exist. Figure 20 elaborates on this problem.

5. We included a variable BELONG_TO_A_PATTERN for every system call in the input file, to avoid re-processing it while looking for the never-seen-before system call. Meaning that a never-seen-before system call should be a system call that does not belong to a pattern and was not previously processed.

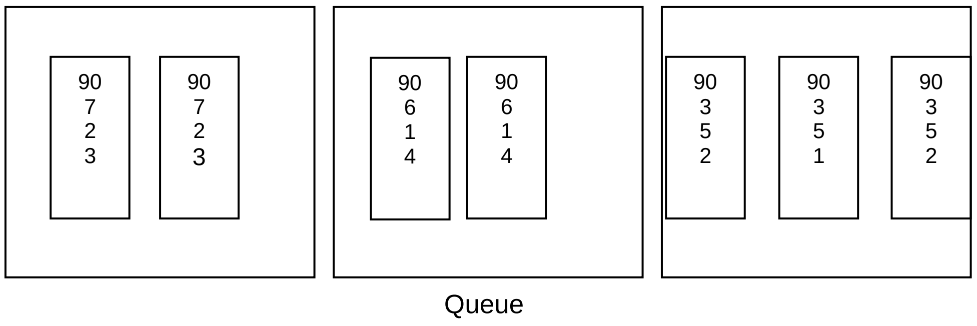
6. Patterns are stored in a hash table pointing to sequences of different lengths as shown in Figure 5.7.



(a)



(b)



(c)

Figure 5.6. (a) different subsequences starting with system call 90 can be expanded concurrently. (b) Similar subsequences are grouped together. (c) Each group of sequences are placed on a queue and processed until all subsequences are examined.

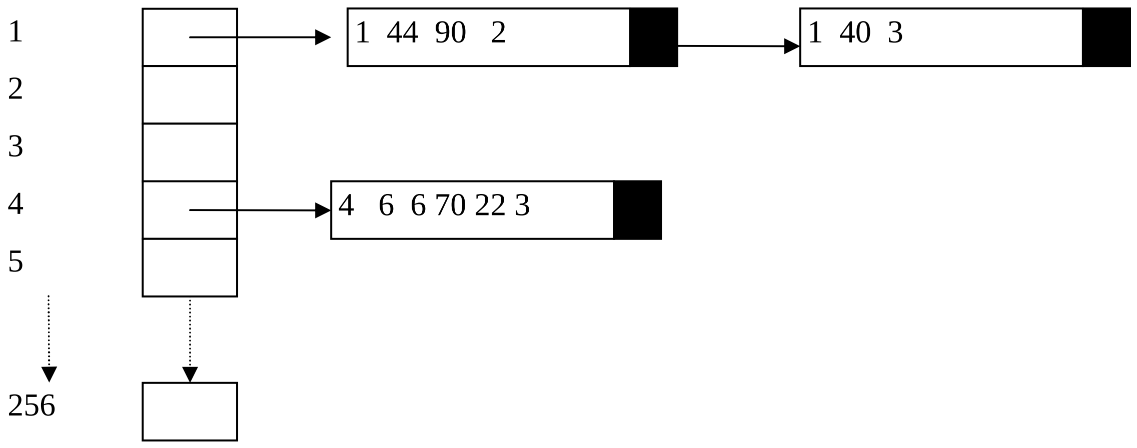


Figure 5.7. Hash table storing Variable-Length with Overlap- Relationship Profile Method entries. Each subsequence is of a variable size.

7. In some cases a sequence will be matched with a pattern in the normal data base because it starts with a specific system call. However, as we continue matching the rest of the string, a number of consecutive mismatches may start to accumulate. If this exceeds a threshold value, for example, 2 or more, then our system will automatically go back to the first system call causing the mismatch and match it against another appropriate pattern. In such a case we backtrack to make sure that the previously accepted system calls do not cause mismatches especially if they did not completely match the sequence. For example, if we are testing the following sequence {4, 1, 40, and 3} against the database in Figure 5.7., we find that the system call 4 has an entry in the database. However, as we test the remaining system calls, a number of mismatches will start to accumulate. If we chose in our system to start checking other patterns if two consecutive mismatches happen, then we will backtrack to system call 1 in our input training file and then check its entry in the pattern database. In this case the sequence {1, 40, and 3} does exist and only one

mismatch is identified which is for the system call 4. The sample code of our implementation of variable-length-with-overlap-relationship method can be found in Appendix D and Appendix I shows the generated patterns.

5.5.3. Performance

The variable length with overlap relationship intrusion detection system was run on the datasets obtained from the website of the University of New Mexico (http://www.cs.unm.edu/~immsec). The login data set was used to detect Trojan horse attacks. Since the patterns generated with this method are variable length, they don't depend on a window size and require only one run to the program. The data obtained from such a run is presented in table 5.10. In general the total time to perform testing was less than 0 seconds. 32 patterns were generated holding 340 system calls. The longest pattern had 43 system calls whereas the shortest pattern had only 2 system calls. Space cost while running is large because it needs to include the size of the hash table, the size of the data structure to hold the information of the pattern itself, pointers, etc.

It is important to explain what is meant with a mismatch in our variable length intrusion detection system. The number of mismatches indicates how many individual system calls were out of place or did not exist in the correct position in their corresponding or appropriate pattern. This is why a large number of mismatches resulted from our run to the program. This is different from flagging a complete sequence as anomalous as in sequence method.

	parameter length
Total testing time (seconds)	0
maximum length	43 system calls /pattern
minimum length	2 system calls / pattern
Average length	22 system calls / pattern
Size of normal DB	32 Patterns
Number of system calls in Normal DB	340
Size of testing input	1350 system calls
Space cost while running (bytes)	206848
Space cost while saved to disk (bytes)	1360
Number of mismatches flagged	766 patterns
% mismatches threshold	56.74 %

Table 5.10. Variable length with overlap relationship method performance.

5.6. Comparison

All methods were able to detect the Trojan horse attack obtained from the website of the University of New Mexico (http://www.cs.unm.edu/~immsec). Table 5.11 compares the three methods. All methods finished testing in less than a second. Window size 9 for both sequence and lookahead pairs methods was chosen for this comparison because it is the recommended in [Somayaji 2002] which is best for storing the sets in 8 bits = 1 byte. Although the number of patterns in the variable length method is much less than the remaining other two methods, the total space cost of maintaining such patterns is very high especially while running the program. However, it requires the least space while saved to disk. Each method identifies mismatches in a different way. In general, the threshold used to raise alarms can be adjusted around the mismatch %. This is equal to the total number of mismatches identified divided by the total number of sequences in the testing file. Both sequence and lookahead-pairs methods were run on the same training and testing sets while varying the window size from 2 to 32 system calls.

	Sequence method (window = 9)	**Lookahead pairs (window = 9)**	**Variable length**
Total testing time (seconds)	0	0	0
maximum length	9 system calls/ sequence	8 pairs / sequence	43 system calls /pattern
minimum length	9 system calls /sequence	6 pairs / sequence	2 system calls / pattern
Average length	9 system calls /sequence	7 pairs / sequence	22 system calls / pattern
Size of normal DB	359 sequence	2724 pairs	32 Patterns
Number of system calls in Normal DB	3231	5448	340
Size of testing input	1342 sequences	10764 pairs	1350 system calls
Space cost while running (bytes)	91908	65536	206848
Space cost while saved to disk (bytes)	12924	10896	1360
Number of mismatches flagged	136 sequences	365 pairs	766 patterns
% mismatches threshold	10.13 %	3.39 %	56.74 %

Table 5.11. performance comparison of sequence method with a window size = 9, lookahead-pairs method with window size = 9 and variable length with overlap relationship method.

The % mismatch threshold is equal to the number of mismatches flagged divided by the total number of sequences processed while testing. For the sequence method if one system call within a pattern caused a mismatch then it is flagged as an anomaly. .

Figure 5.8. shows the space cost of the three systems while running. We can observe that the variable-length with overlap- relationship profile method does not require a window size and therefore, the size of the database is the same no matter how many times we re-generate the training profile. However, both sequence and lookahead-pairs methods are affected by the window size which is considered one of their drawbacks. As observed in Figure 5.8., variable-length-with overlap relationship method will always have the same space cost. Below window size = 6 both sequence and lookahead pairs methods had similar space cost. However, as the window size increases, lookahead pairs starts to have better space cost. At window size = 15, the variable length method starts to have better space cost than the sequence method and at window size = 24 it starts having better space cost than the lookahead pairs method.

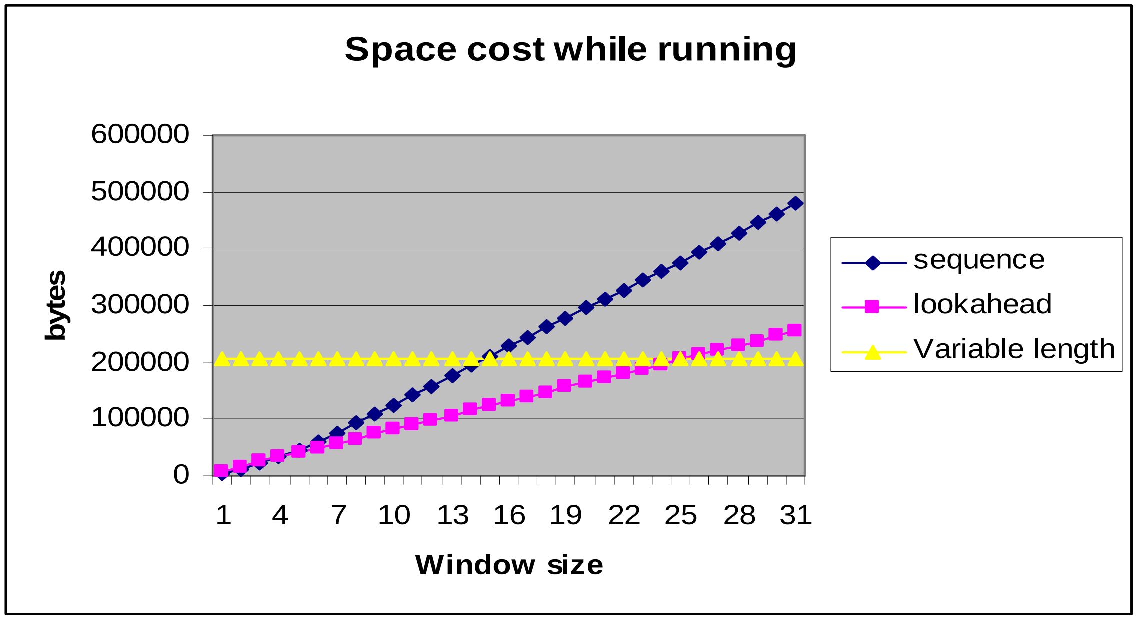


Figure 5.8. Space cost while running of the structures holding the normal pattern DB entries of sequence, lookahead and Variable-Length with Overlap-Relationship Profile Methods.

The order of inputting the training data files does not affect the produced database profiles for both sequence and lookahead methods but does affect the generated database profile for the variable-length method. This is because we are scanning files for the never-seen-before system call and accordingly the first system call in the first file is our first choice. The database patterns generated are slightly different but the detection rate of intrusions is not affected.

We identify “system-call-denial-of-service-attack” as malicious code that can repeatedly call the same sequence of system calls indefinitely. We noticed that in the training input files available at (http://www.cs.unm.edu/~immsec), system call 90, in one file, has been repeated for example more than 10 times repetitively. Furthermore, even if the preprocessing step is avoided and the complete sequence is processed, there is no data structure for holding how many times such system-call was repeated or could be maximally repeated. For example, for both the sequence and the lookahead pairs methods, if the sequence {90,90,90,90,90,90,90,90} was accepted for processing, the following database pattern entries will be generated for a window size of 4.

90 90 90 90

90 90 90 90

90 90 90 90

90 90 90 90

90 90 90 90

Rows 2 to 5 will be deleted because they are repeated and only one will be added to the normal database.

All variable-length-based IDSs developed [Jiang, Hua and Oh 2003] [Jiang, Hua, and Sheu 2002] [Wespi et al. 1998] [Wespi, Dacier and Debar 1999] [Wespi, Dacier and Debar 2000] perform aggregation to identical consecutive system call IDs. This means that there systems cannot defeat the system-call based-denial-of-service attack. However, if the step of aggregating identical system call IDs is removed, the system can generate one pattern that holds such information and its length is equal to the number of times this system call is repeated. For example for the input training sequence {1 2 3 90 90 90 90 90 90 90 1 2 3 1 2 3} the patterns generated with aggregation will be {{1 2 3}, {90}}. However, without aggregation the patterns generated will be {{1 2 3}, {90 90 90 90 90 90 90}}. It is difficult, however, if the system is not fully trained on all possible execution paths to predict - especially for some system calls - what is the allowable times it can be consecutively called. It is worth mentioning that accepting such a long pattern sequence may not be an efficient solution since such system calls may be called less or more times and still be normal. Therefore, it is important to identify such system calls and put a maximum allowable number of times to its repetition and accept similar sequences with less lengths. This can be achieved by correctly assigning a repetition threshold value. However, assume we have the following normal sequence in our DB {90 90 90} and the repletion threshold is 3. If we are under a denial of service attack and we are infinitely reading the system call 90 it will take the system 3 times to read the sequence {90 90 90} before raising an alarm.

5.7. Evaluation

As explained in the previous sub-sections and for off-line testing of the implemented algorithms in this paper, the “login” application was investigated and its input data set for both training and testing were obtained from the website of the University of New Mexico (http://www.cs.unm.edu/~immsec).

In this section we evaluated the system against another dataset for the "ps" application obtained from the website of the University of New Mexico (http://www.cs.unm.edu/~immsec). The traces of the "ps" application contained two log files for normal behavior a stricter test of Trojan horse attack was designed by the University of New Mexico called "home-grown".

Data input pre-processing was conducted on these files to make them suitable for processing. In general, each log consists of the traces of multiple processes interleaved in the same log file. In general, each line in the log file listed pairs of numbers. The first number is the PID of the executing process and the second number is the ID of the system call. We performed pre-processing to the two log files to group entries of the same PID together in separate files. As a result, the first log contained 8 traces and the second contained 15 traces. The system calls associated with every process where grouped in a separate file maintaining the order of system calls. All traces were used to train the implemented systems and to generate their pattern databases. The "home-grown" log file was used to test the systems.

5.7.1. Sequence Method

Table 5.12. show the results obtained when running the sequence method IDS on the "ps" application dataset. As the window size (number of system calls per window) increases, the number of sequences to be stored in the normal database increases. The number of sequences in the normal DB is obtained after removing redundant entries. The number of sequences depends on the number of traces collected in normal behavior, how many subsequences are similar and the window size chosen. For example, if a sequence is repeated several times in the normal database, the initial database will hold many repeated entries and the final database will be smaller.

Suppose we have the following sequence repeated 10 times in the normal training sequence: {90, 5, 20, 100, and 64} with a window size w = 2, a single instance of this sequence will result into 4 patterns in the initial normal database. Since we have 10 instances of this sequence then the total number of entries in the normal database will be equal to 10 * 4 = 40 entries. After removing redundant entries in the database, the reduced normal database will contain only 4 entries or patterns. Therefore, the finial size of the normal database depends on the following:

- The size of the original training dataset.
- The number of repeated sequences within this dataset.
- The window size used to generate patterns.

These elements differ from one trace to another and from one monitored application to another. In table 5.12., the space cost of the normal database while running and while being saved to disk increases as the window size increases since it is proportional to the number of sequences in the normal database. The number of sequences tested decreases by one system call as the window size increases because repeated entries are not removed but they are processed in an online fashion (they are processed as they are read). The mismatch % threshold is equal to the total number of mismatches read divided by the total number of sequences generated from the testing log file while testing.

# sys calls / w	# sequences in normal DB	space cost while running (bytes)	space cost while saved to disk (bytes)	# sequences in testing	Mismatch % threshold
2	55	1764	440	4339	0.89
3	78	4996	936	4338	2.37
4	96	9220	1536	4337	3.38
5	109	13956	2180	4336	4.40
6	125	20004	3000	4335	5.306
7	142	27268	3976	4334	6.206
8	158	35396	5056	4333	7.108
9	175	44804	6300	4332	8.12
10	192	55300	7680	4331	9.14
11	209	66884	9196	4330	10.16
12	227	79908	10896	4329	11.18
13	245	94084	12740	4328	12.22
14	263	109412	14728	4327	13.26
15	280	125444	16800	4326	14.33
16	296	142084	18944	4325	15.56
17	312	159748	21216	4324	16.79
18	327	177892	23544	4323	18.019
19	343	197572	26068	4322	19.25
20	359	218276	28720	4321	20.48
21	374	239364	31416	4320	21.73
22	388	260740	34144	4319	22.99
23	402	283012	36984	4318	24.17
24	416	306180	39936	4317	25.27
25	430	330244	43000	4316	26.36
26	443	354404	46072	4315	27.46
27	456	379396	49248	4314	28.55
28	469	405220	52528	4313	29.65
29	482	431876	55912	4312	30.77
30	495	459364	59400	4311	31.524
31	508	487684	62992	4310	32.18
32	521	516836	66688	4309	32.83

Table 5.12. Sequence method performance across several window sizes for “ps” application.

Furthermore, the mismatch threshold increases as the window size increases because the system call causing the mismatch will appear in more sequences in both the tested sequence and the pattern sequences.

Figure 5.9. shows graphically the number of sequences in the normal databases for both “login” and “ps” applications datasets. The number of sequences is obtained after removing redundant entries in the database and while using sequence method IDS. The numbers of patterns of both applications show positive correlation as the window size increases. Both applications should not have the same slope of increase as they contain different data but the graphs indicate that they behave similarly (increase as the window size increases).

Figure 5.10. and Figure 5.11. show the space cost while running and while saved to disk for both “login” and “ps” datasets. We observe the following: space cost while running is greater than while saved to disk for both “login” and “ps” applications. This is because while running we are considering the size of the data structures used to store patterns in the normal database. However, we are storing to disk only the integer contents of the patterns. When comparing figures 5.10. and 5.11. against each other we note that the behavior is similar in both figures. Meaning that the cost is increasing as the window size increases.

Figure 5.12. compares the behavior of the number of sequences handled while testing and obtained from the testing log file of both “login” and “ps” application datasets. Both decrease by one system call as 5.13. window size increases by one. The range of each application is different because it depends on the size of the log file used for testing each application. From figure 27 we observe that both applications “ps” and "login" have positive correlation for the mismatch % threshold as the window size increases. The slopes of their increase differ because their data is independent from each other.

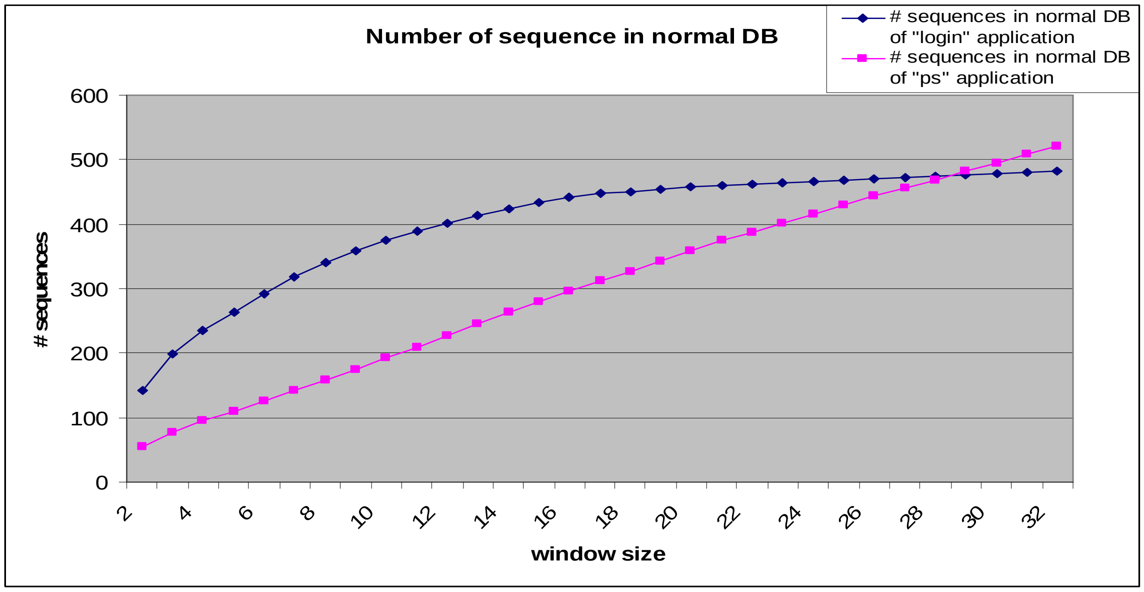


Figure 5.9. number of sequences in normal database for both "login" and "ps" applications datasets. The number of sequences is obtained after removing redundant entries in the database and while using sequence method IDS.

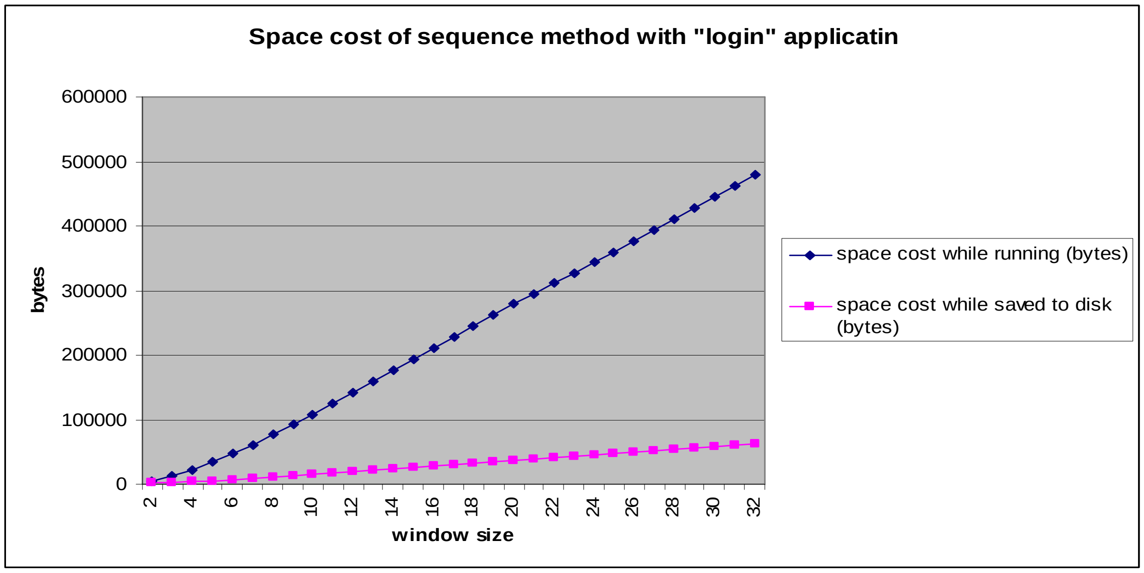


Figure 5.10. space cost of normal database while running and while saved to disk (in bytes) for the "login" application dataset when using the sequence method IDS.

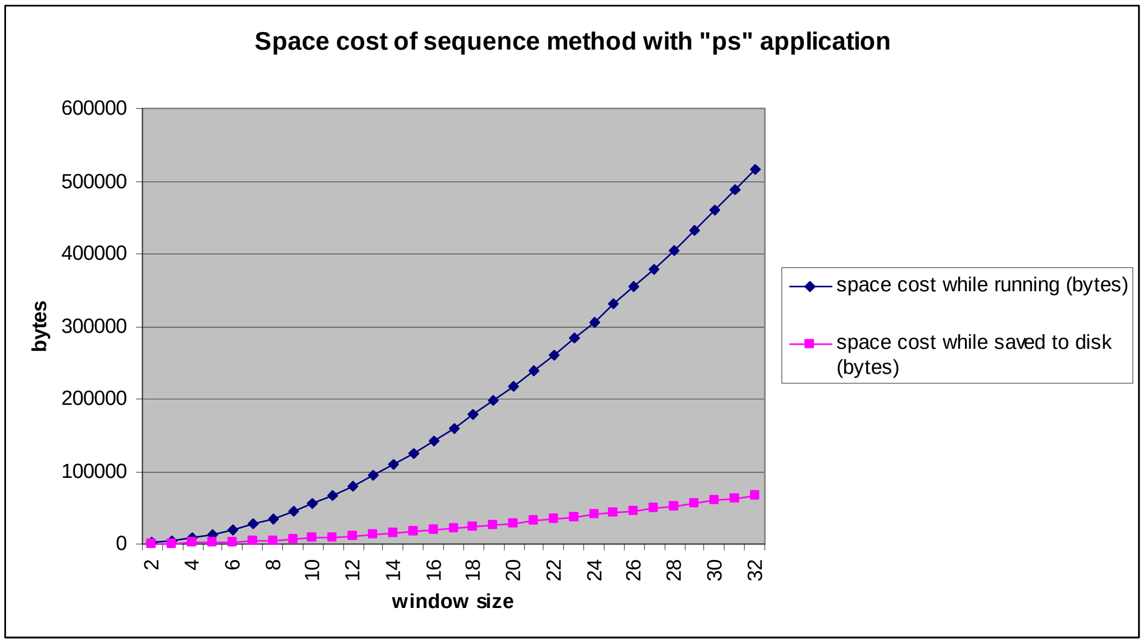


Figure 5.11. space cost of normal database while running and while saved to disk (in bytes) for the “ps” application dataset when using the sequence method IDS.

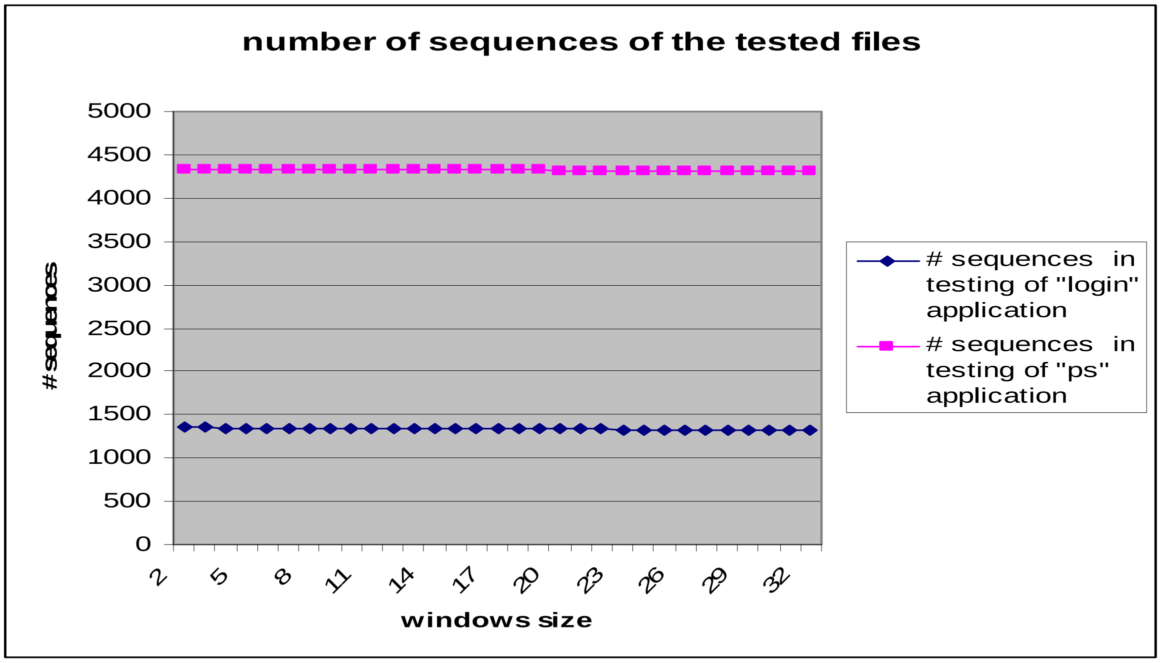


Figure 5.12. number of sequences tested while running the sequence method IDS on both “login” and “ps” applications datasets.

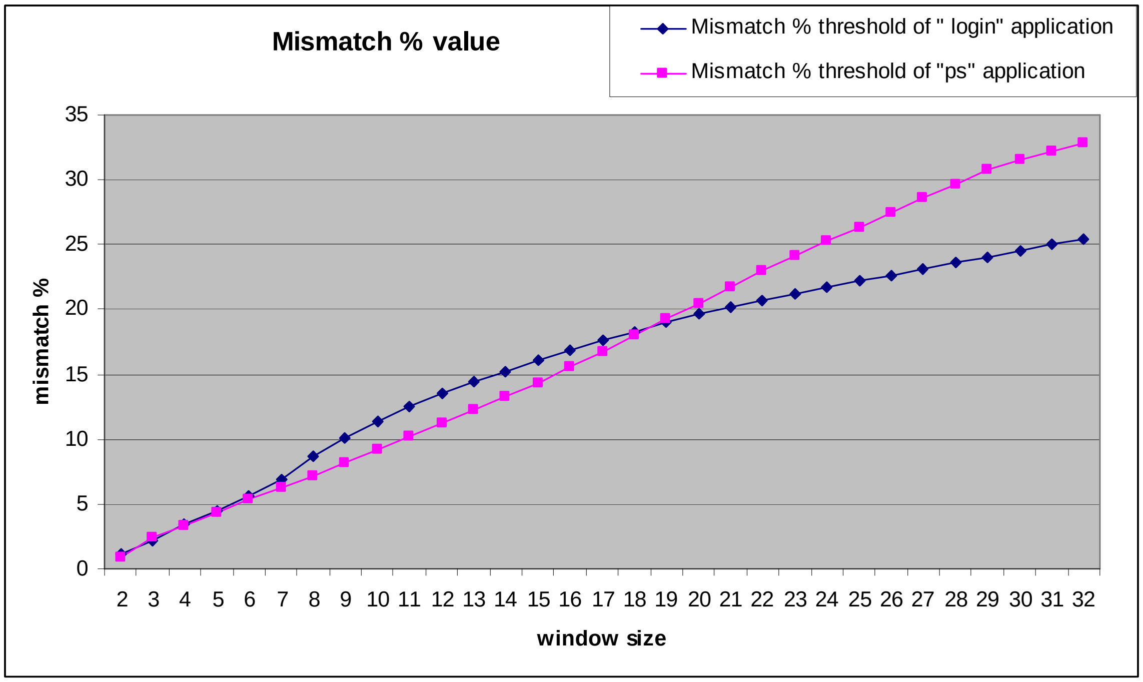


Figure 5.13. mismatch percentage value obtained when testing the "login" and "ps" applications datasets using the Sequence method.

5.7.2. Lookahead-Pairs Method

Table 5.13. shows the results obtained when running the lookahead-pairs method based IDS on the "ps" application datasets. We note that as the window size increases the number of pairs stored in the normal database increases. This number represents the number of patterns kept after removing redundant entries. The space cost while the system is running and while saved to disk is also increasing because it depends on the number of pairs in the normal database. As the window size increases the number of pairs considered while reading the testing log file will increase because we are not removing redundant pairs. The mismatch % threshold is increasing as the window size increases because the mismatching pair appears in more tested sequences as the window size increases.

Figure 5.14. shows the number of pairs of both “login” and “ps” applications when using lookahead-pairs method. The slopes of the lines are different because the data used to generate the pairs are different for each application. This figure show that both lines have a positive correlation with the increase in the window size and that both have the same behaviour.

Figures 5.15. and 5.16. show the space cost while running and while being saved to disk for both “login” and “ps” applications normal datasets. For each application the cost while running is larger than while being saved to disk because while running we are considering the size of all the arrays being used.

However, we are only saving the locations set to 1 while saving to disk. When comparing both figures together we notice that they are positively correlated as the window size increases and that the lookahead-pairs method IDS have the same behaviour for both data sets.

Figure 5.17. compares between the number of pairs resulting from reading the testing log files of both “login” and “ps” applications. There is a positive correlation between the number of pairs and the window size for both applications.

Both applications do not have the same slope because each has its own dataset and the input files for generating the pairs differ and are independent. The number of pairs will be affected by the duration of the testing log file, the number of system calls collected and window size.

Figure 5.18. shows the mismatch % threshold of “login” and “ps” applications when using lookahead-pairs method. Both systems show positive correlation as the window size increases. The systems differ in their slopes since they are independent from each other and are using different log files for both training and testing.

With a smaller window size, “ps” application tends to have a higher mismatch % threshold (i.e. the number of identified mismatches divided by the total number of pairs in the testing log file). However, around window size 9 the behaviour of the “ps” dataset tends to increase.

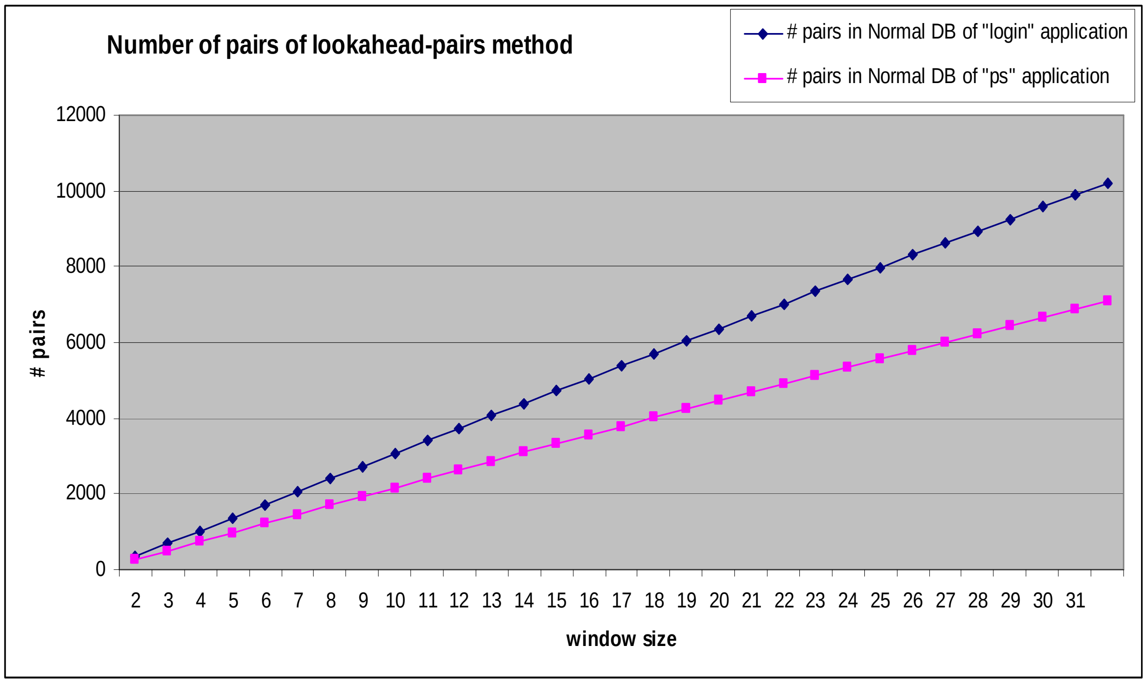


Figure 5.14. Number of sequence in normal database for both “login” and “ps” applications datasets. The number of sequences is obtained after removing redundant entries in the database and while using lookahead-pairs method IDS.

# sys calls / W	# pairs in Normal DB	space cost while running (bytes)	space cost while saved to disk (bytes)	# pairs in testing	Mismatch % threshold
2	244	8192	976	4339	0.898
3	478	16384	1948	8677	1.52
4	729	24576	2916	13014	1.805
5	970	32768	3880	17350	1.86
6	1210	40960	4840	21685	1.77
7	1449	49152	5796	26019	1.93
8	1687	57344	6748	20253	1.841
9	1924	65536	4696	24684	1.92
10	2160	73728	8640	39015	1.99
11	2395	81920	9580	43345	2.09
12	2629	90112	10516	47674	2.215
13	2862	98304	11448	52002	2.33
14	3094	106496	12376	56329	2.499
15	3325	114688	13300	60655	2.710
16	3555	122880	14220	64980	2.899
17	3784	131072	15136	69304	3.066
18	4012	139264	16048	73627	3.271
19	4239	147456	16956	77949	3.491
20	4465	155648	17860	82270	3.733
21	4690	163840	18760	86590	3.951
22	4914	172032	19656	90909	4.14
23	5137	180224	20548	95227	4.368
24	5359	188416	21436	99544	4.591
25	5580	196608	22320	103860	4.79
26	5800	204800	23200	108175	5.005
27	6019	212992	24076	112489	5.196
28	6237	221184	24948	116802	5.376
29	6454	229376	25816	12114	5.52
30	6670	237568	26680	125425	5.655
31	6885	245760	27540	129735	5.786
32	1099	253952	28396	134044	5.919

Table 5.13. Lookahead-pairs method performance across several window sizes for "ps" application.

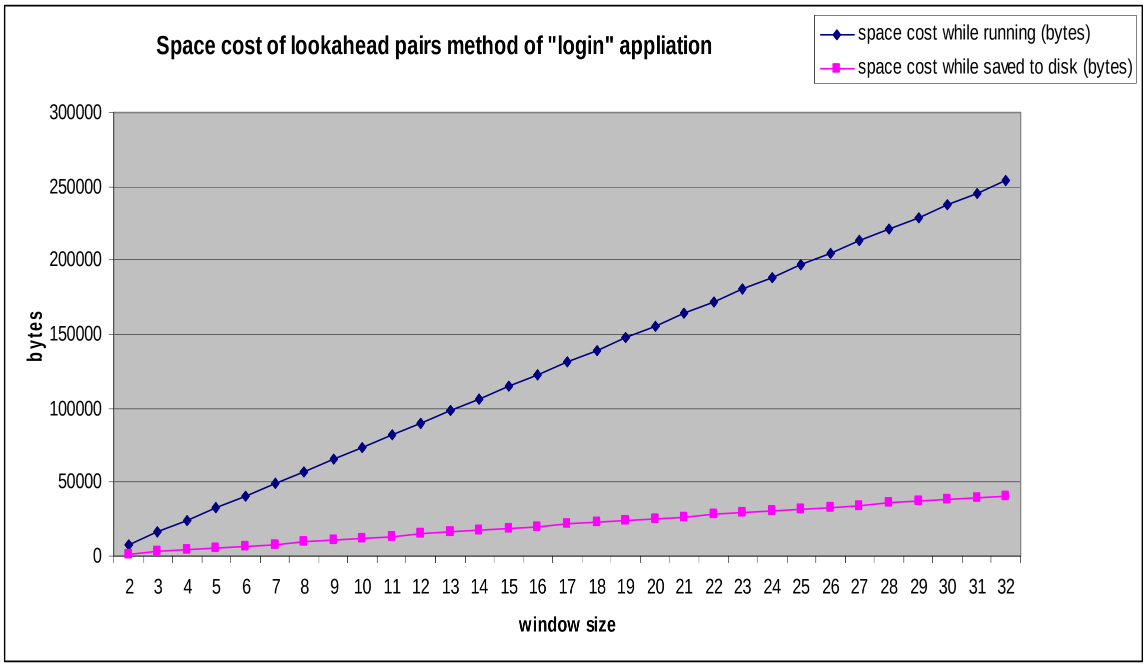


Figure 5.15. Space cost of normal database while running and while saved to disk (in bytes) for the "login" application dataset when using the lookahead-pairs method IDS.

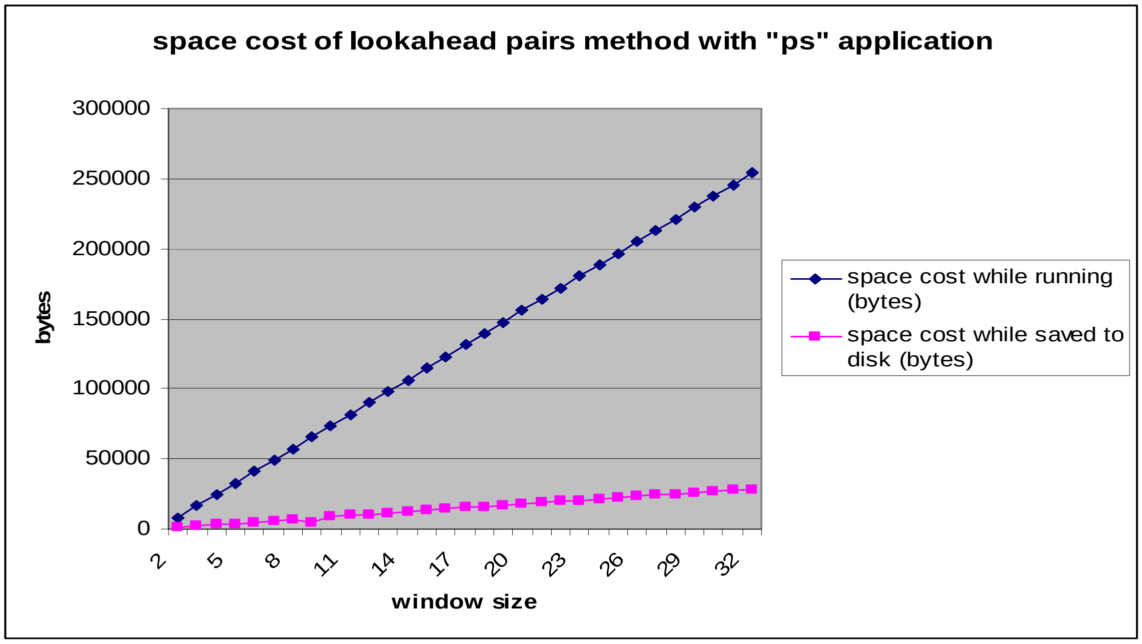


Figure 5.16. Space cost of normal database while running and while saved to disk (in bytes) for the "ps" application dataset when using the lookahead-pairs method IDS.

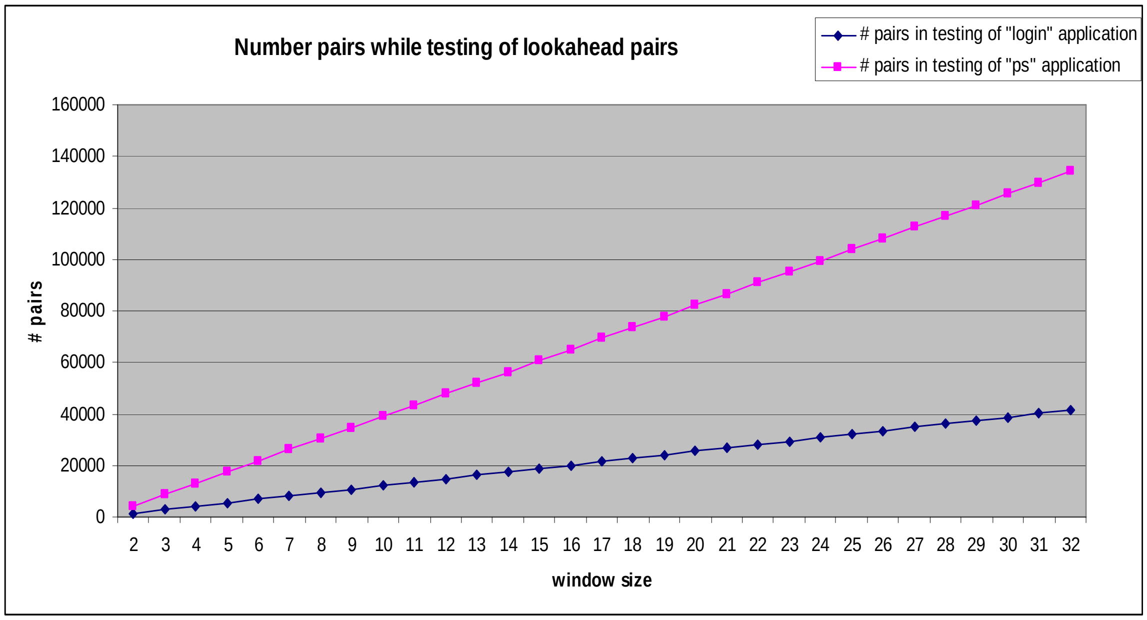


Figure 5.17. Number of pairs tested while running the lookahead-pairs method IDS on both "login" and "ps" applications datasets.

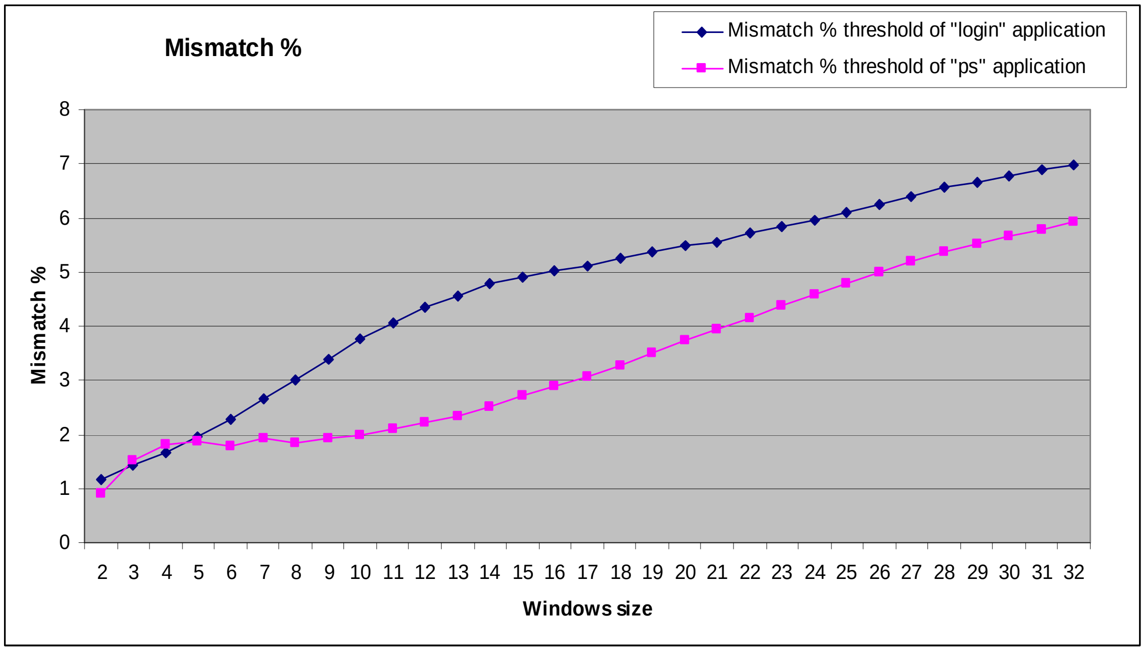


Figure 5.18. Mismatch percentage value obtained when testing the "login" and "ps" applications datasets using the lookahead-pairs method.

5.8. Validation and Verification

Verifying the implemented systems was incrementally performed. Each subsystem was tested against carefully designed test cases that cover all possible input parameters and execute all possible paths of execution. For example, the sequence method based IDS required implementing a subsystem for reading the input log files and processing them to be suitable for further processing. The log files were read and then divided according to PID (parent process ID) to several files. Each file holds the system calls associated with a specific PID. First, on smaller sized files we tested the generated files manually to make sure that the system actually and correctly performed its functions. Another subsystem was to read the files and then start creating a database that holds all possible patterns in an array format depending on the input parameters specified which mainly the window size is. Tracing log files of all of the system activities have been generated and are manually investigated to test if the system is performing correctly or not. The array is then scanned to remove redundant rows and a finial array is produced. The patterns in this finial array is then processed and stored in the normal pattern data structure. The data structure for holding such information is a hash table where each entry in the table points to a linked list of all patterns starting with a specific value. To verify that this data structure is holding the correct information, its content is written to a log file and compared with the original array. To verify that testing is working correctly, several test cases have been created with predefined inputs. For example, several files have been created to test if the system will produce the desired output or not. The files, for example, tested the following cases:

- The tested sequence is normal.
- The tested sequence is compared with one pattern (expands across one pattern)
- The tested sequence expands across several patterns and therefore is compared with more than one pattern.
- The sequence contains only one mismatching system call.
- The sequence contains several mismatching system calls.
- The mismatching sequences are compared with one pattern.
- The mismatching sequence is compared with several patterns.

Finally, when the system passed its verification stage the system was tested on real log files. The activities were recorded in a log file and examined to make sure that the system is still performing correctly.

The same techniques were used to verify lookahead-pairs method IDS. Here the data structure used to store the normal pattern database is one-dimensional array behaving as several 2-dimensional arrays. The locations that are set to one were examined and compared with the original array holding all possible pairs. Furthermore, a list of test cases were generated to test if the system will produce the correct and expected out put or not.

For the variable length with overlap relationship method, the verification was incrementally performed. Since it is the most sophisticated and more time consuming system, care was given to each subsystem. The input data for normal behavior was first read to a data structure that consisted of linked list of linked lists (similar to tree structure). This is because extensive processing will be performed on the input data. The content of such lists were checked to make sure that the data was correctly stored. Identifying patterns was an extensive process where we had to re-read and re-process the items in the data structure. Each performed step was recorded in a log file and manually examined to insure correct execution. After generating the patterns and making sure that they are correctly stored in the final normal data base structure, several test cases have been implemented to test matching. Files for testing were created to check matching against one, two or more patterns in the database. Both normal and intrusive sequences have been checked. As a conclusion to this stage, the three implemented systems passed the verification stage.

Three techniques have been used to validate the implemented systems: trace validation, sensitivity validation and graphic displays. Trace validation is similar to verification where a list of test cases has been created and all possible execution paths have been identified. With this list of input cases the output has been observed and the following is a sample of the possible input cases:

- Sequence is normal.
- Sequence contains one mismatch.
- Sequence contains several mismatches.
- The sequence does not exist in the normal database.
- The sequence is checked against only one pattern.
- The sequence is checked against several patterns.

Sensitivity validation was also used. Each system was originally tested against the “login” application dataset as shown and explained previously. To test if the systems pass the sensitivity validation test another dataset was examined. The results obtained from using the “ps” application dataset were shown and explained in Tables 5.12. and 5.13. The results indicate that the implemented systems are not sensitive to the input data set. The third validation technique used is the graphic displays and which is summarized and explained in Table 5.14.

Figures	**Systems compared**	**Applications compared**	**Observations**
5.9.	Sequence method	Login and ps	Number of sequences of normal log files of “login” and “ps” applications has a positive correlation with the increase of window size.
5.10. and 5.11.	Sequence method	Login and ps	The space cost of normal log files “login” and “ps” applications datasets has a positive correlation with the increase in window size.
5.12.	Sequence method	Login and ps	The number of sequences of tested log files of both “login” and a “ps” application behaves in a similar fashion.
5.13.	Sequence method	Login and ps	The mismatch # threshold of the testing log files of “login” and “ps” applications behaves similarly.
5.14.	Lookahead-pairs method	Login and ps	Number of pairs of normal log files of “login” and “ps” applications has a positive correlation with the increase of

			window size.
5.15. and 5.16.	Lookahead-pairs method	Login and ps	The space cost of normal log files "login" and "ps" applications datasets has a positive correlation with the increase in window size.
5.17.	Lookahead-pairs method	Login and ps	The number of pairs of tested log files of both "login" and a "ps" application behaves in a similar fashion.
5.18.	Lookahead-pairs method	Login and ps	The mismatch # threshold of the testing log files of "login" and "ps" applications behaves similarly.

Table 5.14. Explaining the graphic display validation

As shown in table 5.14. the behavior the systems are similar and consistent even when tested against different normal and testing datasets.

5.9. Summary

Three host-based IDSs using system call profiles were examined. We tested sequence method, lookahead-pairs method and variable-length with overlap-relationship profile method. Testing lookahead-pairs method is straight forward. The system call and its previous entries (up to window size) are checked against their corresponding entries in its corresponding array and if it does not exist then a mismatch is raised. For both sequence and variable length methods, as system calls are read they are checked against their entries in the hash table and the closest match in a pattern in that entry are chosen and expanded. If no exact match is found a mismatch is raised. For example, the following case is handled in our implementation. Suppose entry 90 in the hash table contain the following sequence {90 4 8} and entry 105 contains the following sequences {{105 4 27}, {105 18 2}}. Testing the input sequence {18 2} will raise 2 alarms since entry 18 in the hash table is empty. Testing input sequence {105 90 4 8} will raise only 1 alarm, since our system will check the entries of 105 and find that the second system call 90 does not belong to any sequence at the 105 entries. However, we test to see if it belongs to another entry in the hash table, which in this case is true and the remaining subsequence {90 4 8} raises no alarms. Finally, testing the input sequence {90 18 2} will raise 2 alarms since 90 does belong to the hash table and although {18 2} exist as subsequences of another entry they should not be accepted. Our implementation differs from the original papers in:

1. The systems were implemented on Windows-XP Operating System using Microsoft Visual Studio 2005 as a win32 console application rather than using gcc compilers on UNIX and LINUX Operating Systems.
2. Normal patterns of both sequence and variable length with overlap relationship methods were stored in a hash table where each entry is pointing to the sequence patterns.
3. Type 2 of variable length with overlap relationship was expanded to handle the sub-grouping of sequences and furthermore sub of sub-grouping if exist.

The following were concluded:

1. All methods can not defeat system-call-denial-of-service attack.

2. Input order of training files does not affect the final constructed databases for sequence and lookahead-pairs methods but affects the variable-length patterns method.

3. Keeping the same number and content of input training files for the variable length method constant but changing the order and in specific the first system call handled will result in different constructed database but will not affect the number and type of intrusions detected.

In general, we observed the following:

T1: {90, 4, 7, 1}, T2: {4, 7, 2}, T3: {4, 7, 2, 90, 115}

If we start our training with T1 and system call 90 the following pattern set will be generated:

{{90, 4, 7, and 1}, {4, 7, and 2}, {90, 115}}

However, if we start with T2 and system call 4 the following pattern set will be generated:

{{4, 7}, {90}, {1}, {2}, {2, 90, 115}}

4. Lookahead pairs method had the best space cost while running, as long as its window size is below 24. As the window size increase, the space cost of storing the associated database also increases and variable length method starts to have a better space cost.

5. In order to investigate whether the output or the behavior of the sequence method and lookahead-pairs method IDSs performs similarly with other input datasets, the two systems will also be tested with “ps” application datasets. As shown in the

previous section, the output parameters gathered for both systems performs in a consistent manner when using both datasets.

Finally, each technique used in this dissertation to implement the associated intrusion detection system has its benefits and drawbacks. All techniques experience better detection as the window size increases especially for sequence and lookahead-pairs methods. Variable length method produces better coverage and is considered a logical representation to patterns. However, it is very expensive to generate the patterns and require high storage requirements. In this dissertation we decided, after experimentation, to continue with the data structure used with lookahead-pairs method. It is easy to manipulate and access. With regard to space cost it had the best storage requirements and it is very fast and easy to access the elements of its data structure.

CHAPTER 6

A DANGER THEORY MODEL

This chapter proposes a danger theory model for intrusion detection. Danger theory can be applied to various application areas. Among these, intrusion detection is the most closely linked to the human immune system. The literature survey presented in this dissertation demonstrates attempts to build artificial immune models based on innate and adaptive immunity. These artificial immune systems implemented a generic architecture to model immunity response in general. They have incorporated both innate and adaptive immunity concepts and built a framework where, after specifying a list of input parameters, different components of the immune system may interact.

In this chapter, we focus on developing a model to represent the danger theory interactions of an adaptive immune system. The danger theory model is designed for distinguishing normal activities from abnormal activities and responding to invasions. The danger theory based intrusion detection system response is governed by the output produced by the antigen presenting cells (APCs).

Dentric Cells (DCs) are one type of APCs and process antigen signatures in their context. The system operation is divided into 2 phases: Training and testing. In the training phase a database of patterns representing normal behavior is created either using positive or negative selection. B cells and DC cells are assigned patterns by which it can detect abnormal bacteria or intrusion signatures.

In the testing phase, monitored system call sequences are scanned by B and DC cells. DC also senses danger signals, if any, and present the bacteria signature in context to T helper cells for further processing and handling.

Figure 6.1. explains the primary immune system response to the presence of bacteria and danger signals. In Figure 6.2., the flow chart of the primary immune system response is presented.

In general, B cell captures antigen (specific antigen) and at the same time DC captures antigen (any antigen) and senses danger signals.

DC presents the antigen in context (antigen signature and surrounding status: natural or un-natural death) and causes naïve T cell maturation. T cell, accordingly, differentiates to T helper 1, T helper 2 and T killer cells.

B cell presents antigen to Th1 and then Th1 primes or downgrades killer T and Th2 cloning. Depending on the strength of the prime message, Killer T and Th2 start their cloning expansion and generation of memory killer T, memory Th1 and memory Th2.

Th1 confirms the antigen to B cell and B cell starts to secrete antibodies that bind to antigen to flag it for destruction. B cell starts its cloning expansion and generation of memory B cells. After B cell binds to the bacteria, the cloned Killer T cells starts attacking the previously flagged bacteria. It is important to understand that only the bacteria that was flagged for destruction will be eliminated by the Killer T cells. At the same time other non-dangerous bacteria may exist in the same zone and will not be killed.

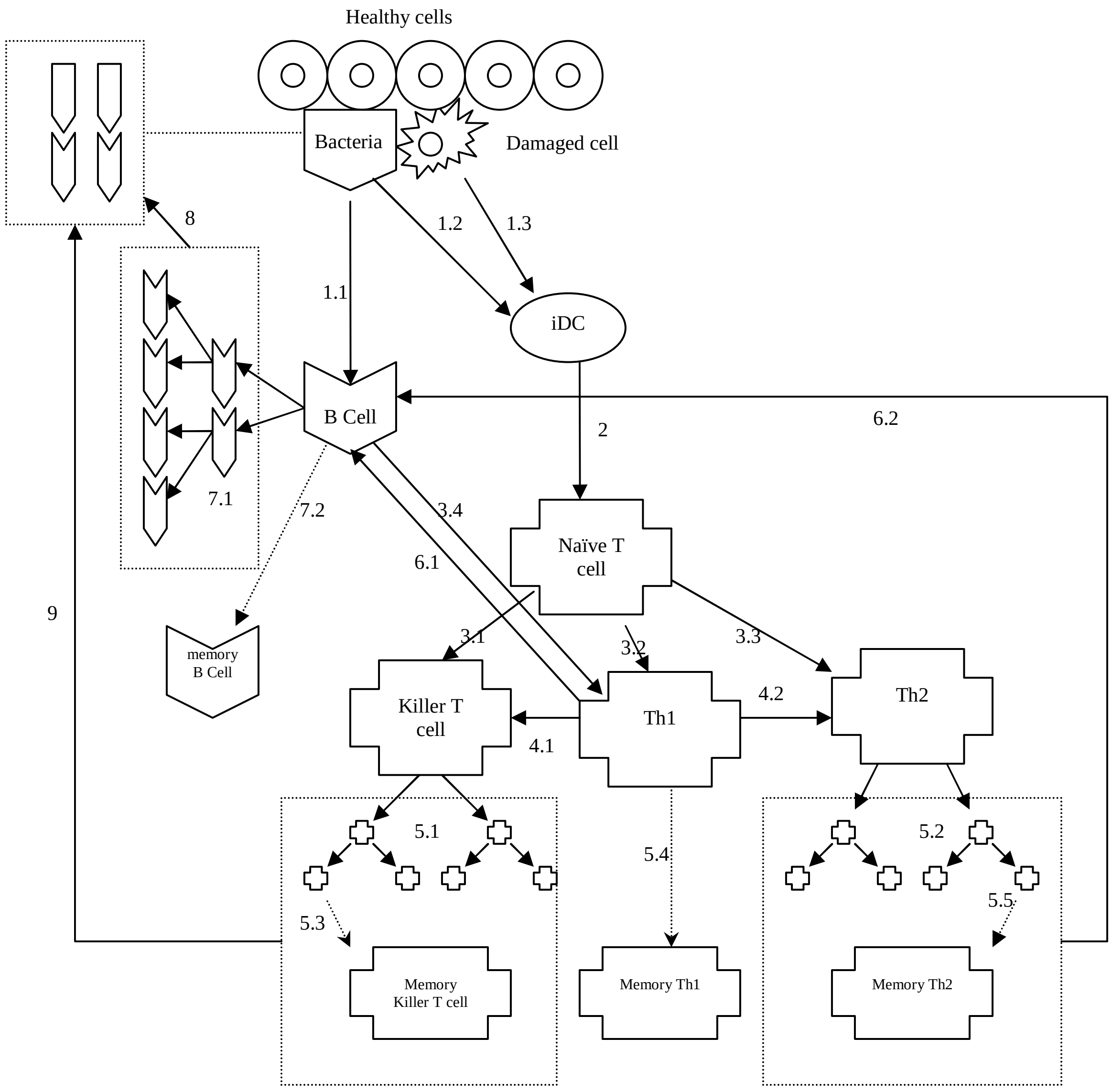


Figure 6.1. Primary immune system response.

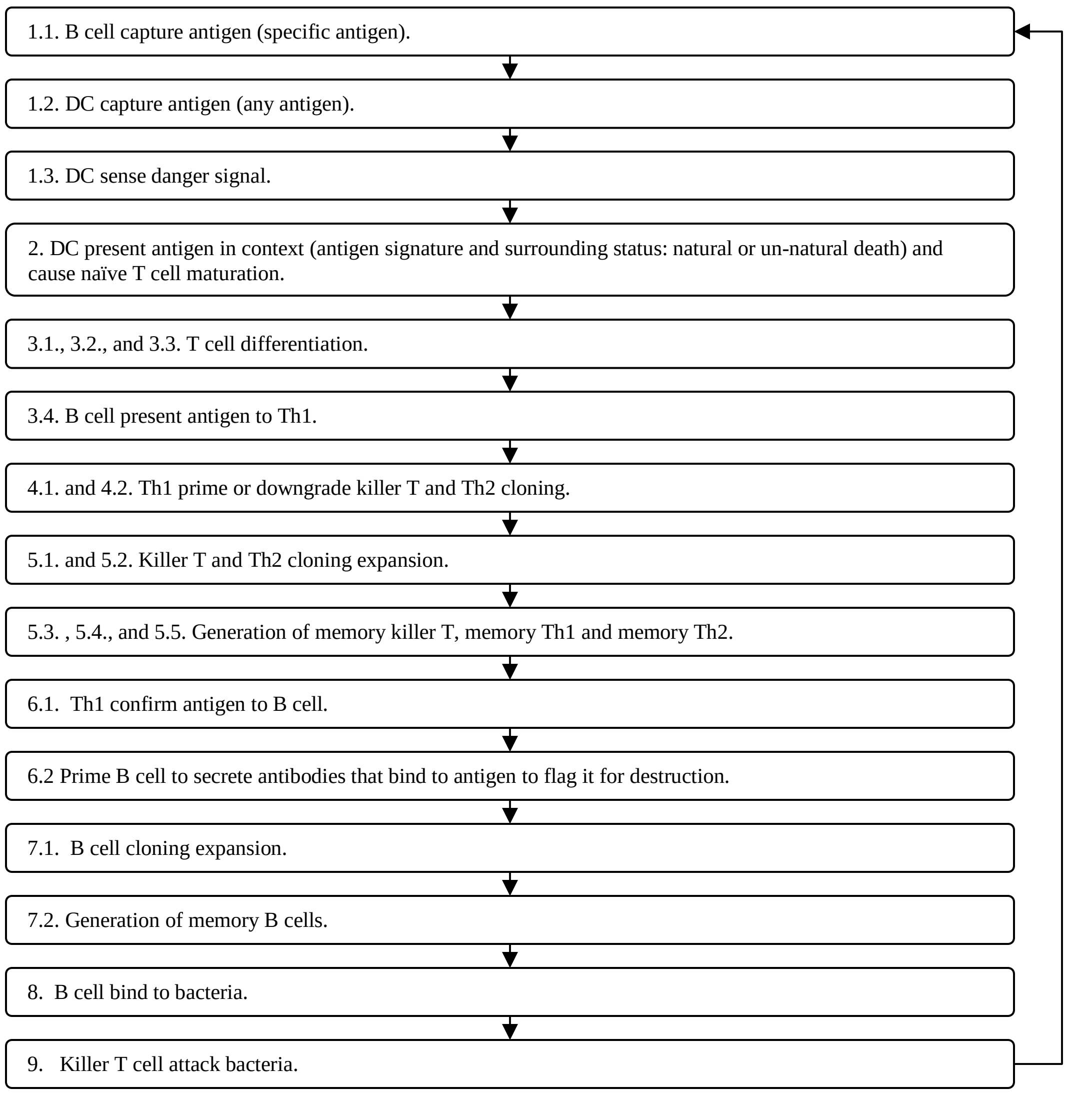

1.1. B cell capture antigen (specific antigen).
1.2. DC capture antigen (any antigen).
1.3. DC sense danger signal.
2. DC present antigen in context (antigen signature and surrounding status: natural or un-natural death) and cause naïve T cell maturation.
3.1., 3.2., and 3.3. T cell differentiation.
3.4. B cell present antigen to Th1.
4.1. and 4.2. Th1 prime or downgrade killer T and Th2 cloning.
5.1. and 5.2. Killer T and Th2 cloning expansion.
5.3. , 5.4., and 5.5. Generation of memory killer T, memory Th1 and memory Th2.
6.1. Th1 confirm antigen to B cell.
6.2 Prime B cell to secrete antibodies that bind to antigen to flag it for destruction.
7.1. B cell cloning expansion.
7.2. Generation of memory B cells.
8. B cell bind to bacteria.
9. Killer T cell attack bacteria.

Figure 6.2. Flow chart of the primary immune system response.

Figure 6.3. and Figure 6.4. explain the secondary immune system response. It is similar to the primary immune system response but differs in the fact that bacteria and sensing danger it now performed by memory cells of both B and DC cells and that such processing is performed faster. This is because in the secondary immune system response, the bacteria have been seen before; therefore the immune system reaction will be specific and faster.

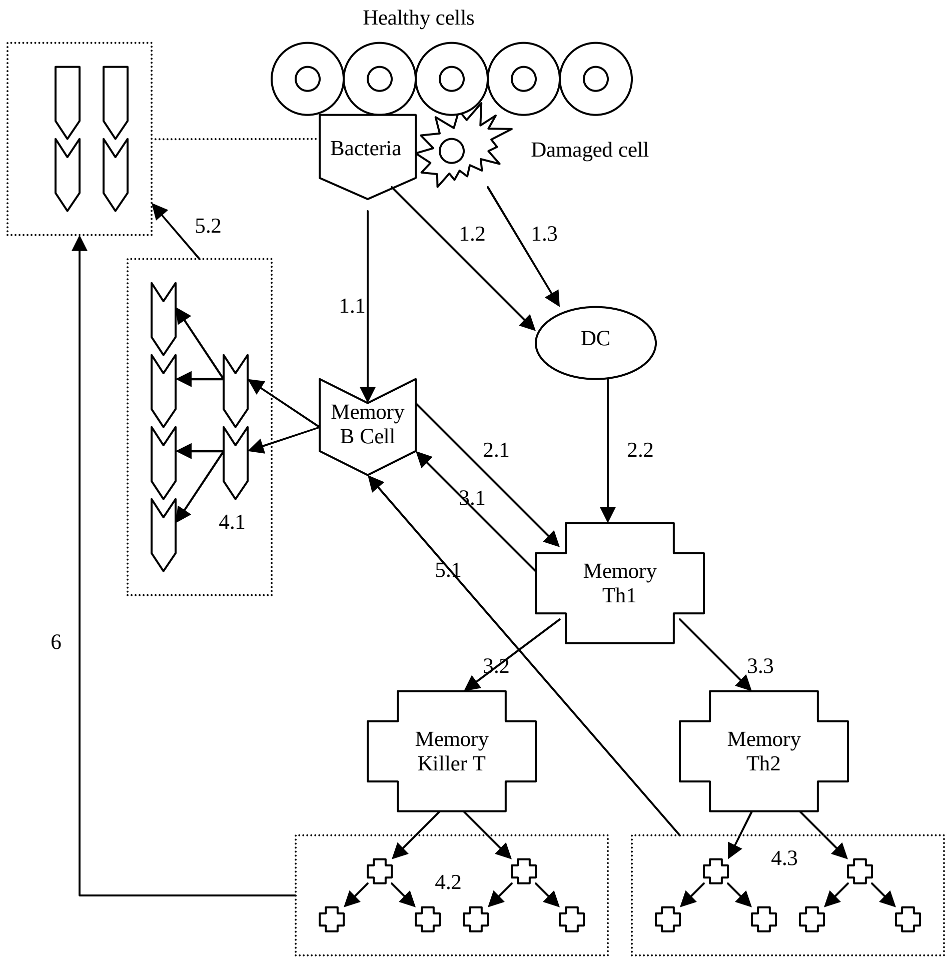


Figure 6.3. Secondary immune system response.

Figure 6.5. explain the life cycle of both B and memory B cells. Each B and memory B cell is given an activation threshold and a life span. B cell circulates the system or the body. If it exceeds it activation threshold then it becomes a memory B cell. On equal intervals, the activation threshold is checked and if it did not reach at least its minimum activation threshold within its life span then it is deleted.

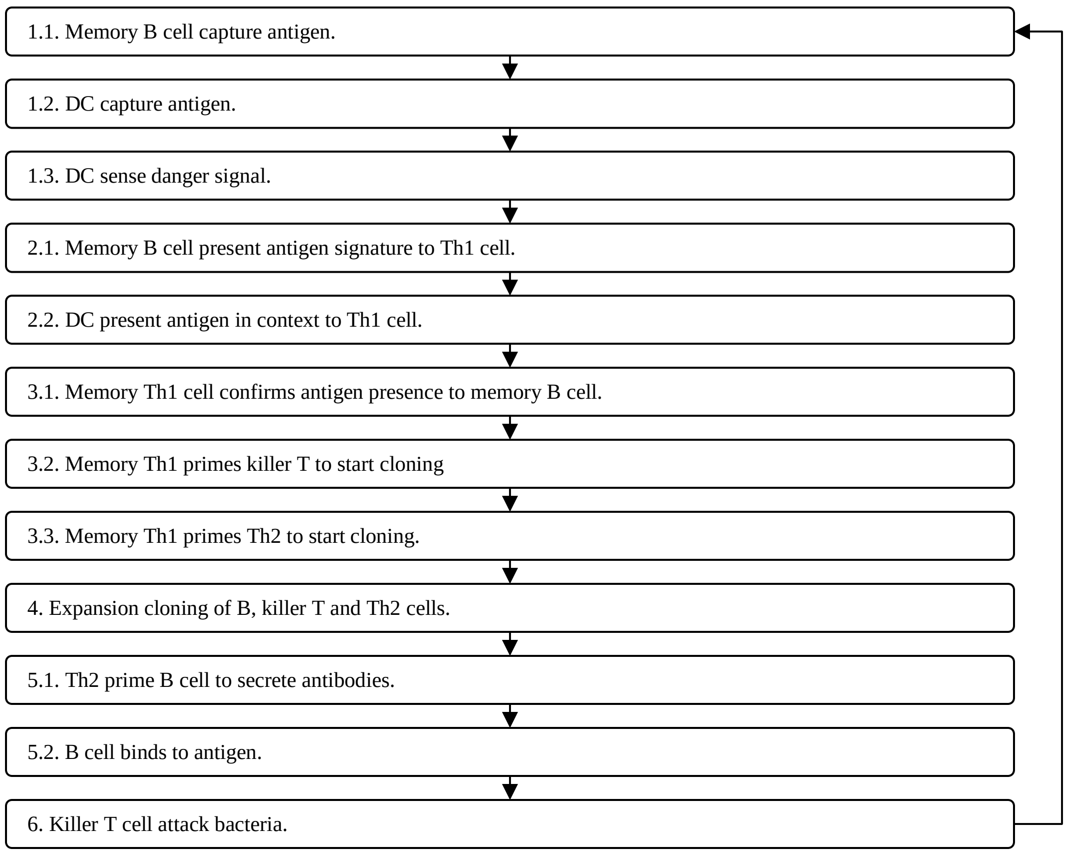

1.1. Memory B cell capture antigen.
1.2. DC capture antigen.
1.3. DC sense danger signal.
2.1. Memory B cell present antigen signature to Th1 cell.
2.2. DC present antigen in context to Th1 cell.
3.1. Memory Th1 cell confirms antigen presence to memory B cell.
3.2. Memory Th1 primes killer T to start cloning
3.3. Memory Th1 primes Th2 to start cloning.
4. Expansion cloning of B, killer T and Th2 cells.
5.1. Th2 prime B cell to secrete antibodies.
5.2. B cell binds to antigen.
6. Killer T cell attack bacteria.

Figure 6.4. Flow chart of the secondary immune system response.

However, if it is above its minimum activation threshold then it is sent to the bone marrow where it undergoes hyper mutation and a new signature is created for the B cell. Memory B cells also circulate the body or the system and on equal intervals their associated activation threshold is checked. If it exceeds it maximum activation threshold it is left to circulate the system. However, if it did not exceed its activation threshold within its life span it is sent to the bone marrow where it undergoes hyper mutation and new signature is created.

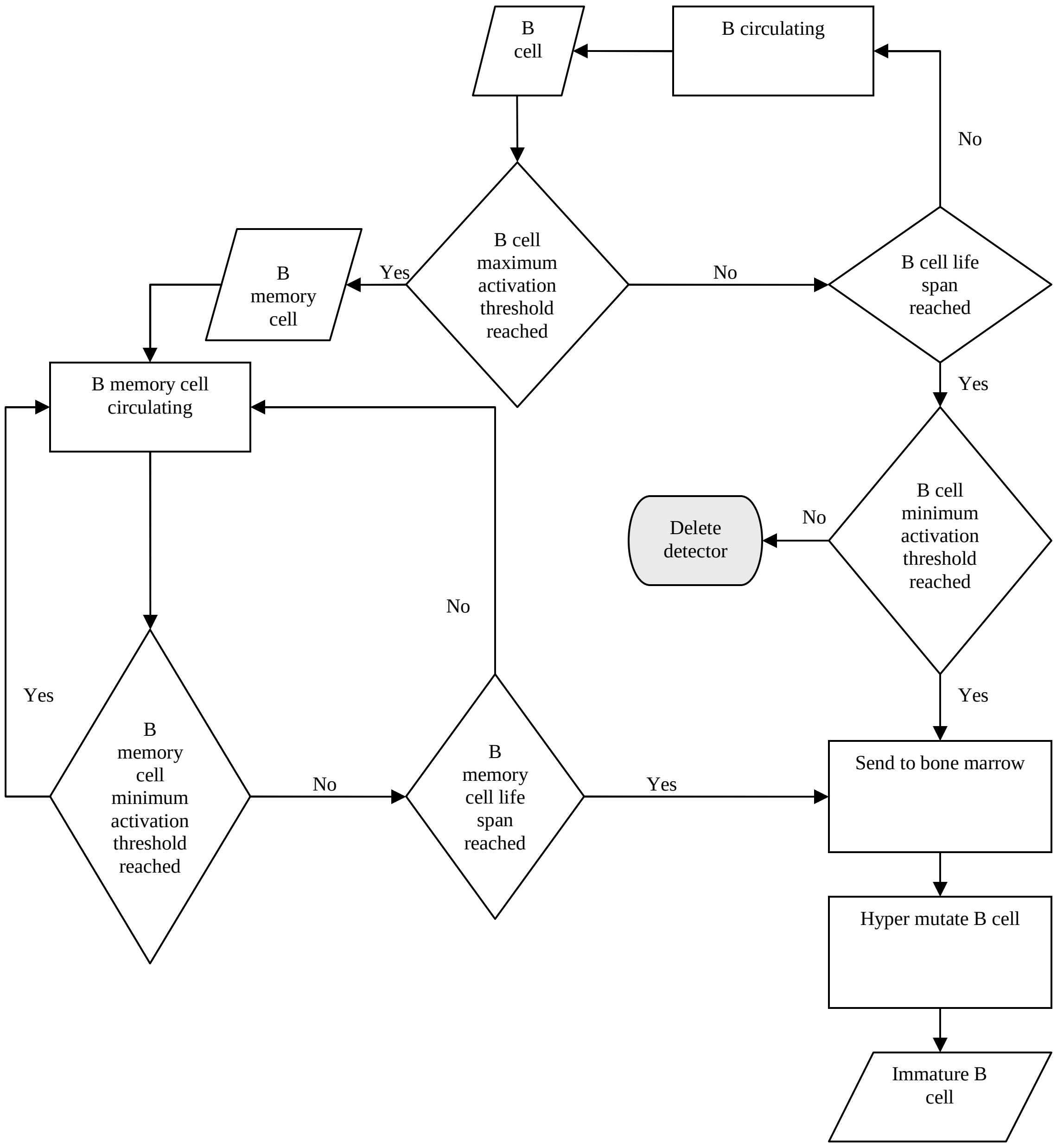

B cell
B circulating
No
B cell maximum activation threshold reached
Yes
No
B memory cell
B cell life span reached
B memory cell circulating
Yes
Delete detector
No
B cell minimum activation threshold reached
No
Yes
Yes
B memory cell minimum activation threshold reached
No
B memory cell life span reached
Yes
Send to bone marrow
Hyper mutate B cell
Immature B cell

Figure 6.5. B and memory B cells life cycle.

CHAPTER 7

ENHANCING LOOKAHEAD-PAIRS METHOD WITH DANGER THEORY

7.1. Introduction

Lookahead-pairs method has been used to implement an AIS based IDS for monitoring system calls. It has outperformed sequence method and overlap-relationship methods in terms of the storage requirements. Due to the smaller storage requirements of lookahead-pairs, it has been chosen in this dissertation for further investigation. In this chapter the lookahead-pairs based intrusion detection system, will be enhanced by incorporating techniques of danger theory. The newly modified systems will be examined and tested. What we mean with lookahead-pairs based intrusion detection system, is an IDS that monitor system calls sequences to find deviations from normal pattern database and uses several 2 dimensional arrays to store the relationships between patterns in the normal database. The database was previously created after being trained on normal behavior with a pre-specified window size. Usually, the current monitored system calls are read and compared with the entries in the database. If a deviation is observed, meaning that it does not exist in the database, a mismatch is flagged.

Lookahead-pairs method is affective in detecting deviations; however, in many cases more advanced intrusion attempts go undetected because they are not only based on system call deviations but also on other parameters such as high CPU and memory usages. Such parameters indicate if the system is under stress or danger. Not only considering the mismatch instances but also other factors and conditions such as signals of the system, is one of the basic concepts of danger theory. Danger theory states that we should not only respond to foreignness (i.e. mismatches) but also to danger signals (i.e. unacceptable system conditions such as high CPU or memory usages).

What we mean with enhancing lookahead-pairs method with danger theory concepts is that we take the main characteristics of lookahead pairs method and then add the functionalities of different danger theory components such as B cell, T cell and iDC cells. The affect of each component on the overall performance will then be examined and measured.

In general, despite the acceptable cost associated with such merge, the performance (i.e. better detection, lower false negative and false positives) is enhanced. This enhancement has been accomplished in three stages. In the first stage the functionality of the APC cell was implemented and tested. APC cell is responsible for detecting mismatches as well as sensing signals (both danger and safe). One type of APC cell is iDC cell which differentiate to either semi mature DC (in case of normal behavior) or mature DC (in case of intrusive behavior). In the second stage, we experimented with generating positive and negative detector sets. A positive detector set is generated by mapping normal behavior to a set of patterns and then stored in the database. Negative detector sets are generated by mapping the complement of normal behavior to a set of patterns. The benefits of each type of set are then examined. The third and final stage is to incorporate all components of danger theory that are B, Th1, Th2, Killer T, iDC and DC classes.

7.2. Experiment Setup

The components of this system are implemented with Microsoft Visual Studio 2005 as a win32 console application. For off-line testing of the implemented algorithms in this paper, two sets of training and testing datasets can be used.

The first was obtained from Danger Theory Project website [http://cs.nott.ac.uk/~jpt]. The datasets provide traces of system calls and associated CPU and Memory usages. For example, “rpc.statd.normal1.tcr.log” which represents a 38 seconds tcreplay log file produced 398 antigens and 9 signals. The following is a sample single line in the database. Personal communication with Twycross, explained the following components of the line entry in the dataset.

```
1137704943.969283  signal  366  4  1  0.00000  1122104  335872
```

Column 1: timesec.timeusec

Column 2: type (in this case this entry in the data set represents a signal entry)

Column 3: process ID

Column 4: NUM signals

Column 5: total number of processes (self + children) (integer)

Column 6: total CPU usage for processes (%)

Column 7: total size of processes (bytes)

Column 8: total size of memory resident portion of processes (bytes).

These signals change over time since there is an interaction (either normal usage or an attack) with the monitored processes (the rpc.statd server and its children), which causes the monitored processes to use different CPU and memory resources.

In order to closely test the performance of our system a second data set was used. Since we are comparing the performance of the enhanced system against our initial implementation of lookahead-pairs method IDS, we decided to use the same datasets used previously and obtained from the website of the University of New Mexico (http://www.cs.unm.edu/~immsec). This file contains traces of system calls generated while running the login application. One problem with this file is that it does not hold values for both CPU and memory usages. This is why we intentionally injected at different time intervals and different locations signal values of CPU and memory usages, creating several test cases to investigate.

Table 7.1. briefly indicates the different attack scenarios that can be handled with the danger theory enhanced system. Our system, furthermore, is not only considering the current situation of the system but also previous state of the system. This is because there might be a normal burst in memory or CPU usage and this should not initiate a response. If, however, such an action (i.e. high CPU, high memory usage, or mismatch) is higher than an acceptable threshold value or has been high for a predefined time range then an intrusion is flagged. Such a decision lowers false positives since the system ignores any deviation of acceptable behavior within limitations. Such deviation results especially when the system is not fully trained on all possible acceptable program execution paths. We are also incorporating the parameters of having the user present or not. Since usually, if there is an activity going on without the presence of a user then this is considered a suspicious activity. We also considered monitoring for the use of abnormal signals that intentionally kill a process.

Type of attack	**Possible condition**
Immediate attempt(s) to perform CPU attack	Yes / No
Immediate attempt(s) to perform memory attack	Yes / No
Immediate attempt(s) to perform mismatch attack	Yes / No
Immediate attempt(s) to use abnormal signals	Yes / No
Presence of user	Yes / No

Table 7.1. attack types that can be identified by a danger theory enhanced IDS.

For example the login datasets were modified to incorporate different situations and the following are some test case examples:

1. In the first scenario there are no mismatches identified and both the CPU and memory levels are normal.
2. In the second scenario, a mismatch was identified but both CPU and memory levels are normal. If previous conditions are normal then no intrusion is flagged. However, if this mismatch is the i^{th} of series and exceeds a predefined acceptable threshold then an intrusion is flagged even if there is no affect on the CPU and memory usages.
3. In some cases either CPU or memory will experience a burst in performance with no mismatches identified. If the previous performance of both have been low and this is the first encounter of such performance then no intrusion is flagged. However, if such high usage of CPU or memory has been noticed and exceeds a predefined period then an intrusion is flagged even if the system did not encounter any mismatch instances. Such an attack can be performed by running a script or a

Trojan horse code that continuously repeat acceptable sequence of system calls indefinitely.

4. In the case that a mismatch and high CPU or memory is noticed, an intrusion is flagged if the current condition as well previous conditions exceed a predefined threshold.

7.3. Lookahead Pairs Method Enhanced With iDC and DC Classes

iDC and DC cells are types of APC cells that gather information from the surrounding environment and act accordingly. The information gathered is mainly the identification of bacteria (i.e. system call sequence mismatch) and sensing signals (i.e. CPU and memory usages).

This does not mean that the iDC cell is only active when there is an attack, but it can also indicate that the system is normal to other cells in the body such as T cells. After identifying bacteria and sensing signals iDC differentiate and start secreting either mature or semi-mature cytokines indicating an intrusion or normal behavior respectively.

These cytokines control the behavior of immature T cells that differentiate to helper T1 (Th1) cell, helper T2 (Th2) cell and Killer T cells. Figures 7.1. and 7.2. explain the general procedure of iDC and immature T cell differentiation.

After Th1 cells are responsible for controlling or managing the behavior of other T cells such as Th2 and killer T cells. It is responsible for priming or suppressing their behavior and cloning expansion degree. If, for example, Th1 is priming killer T cell, then the degree of cloning will be much higher than if Th1 was suppressing killer T cell.

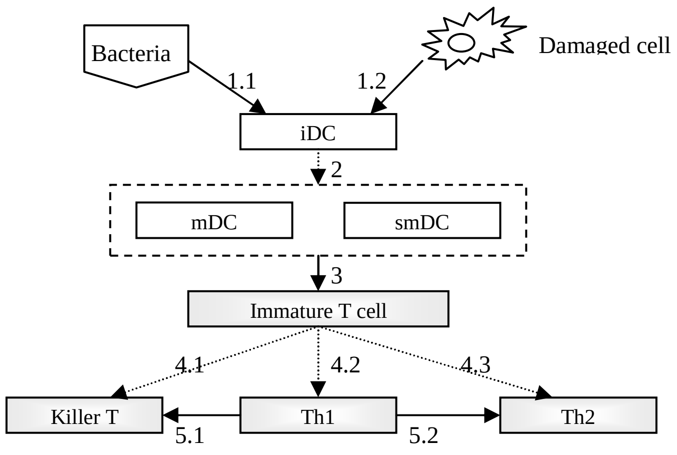
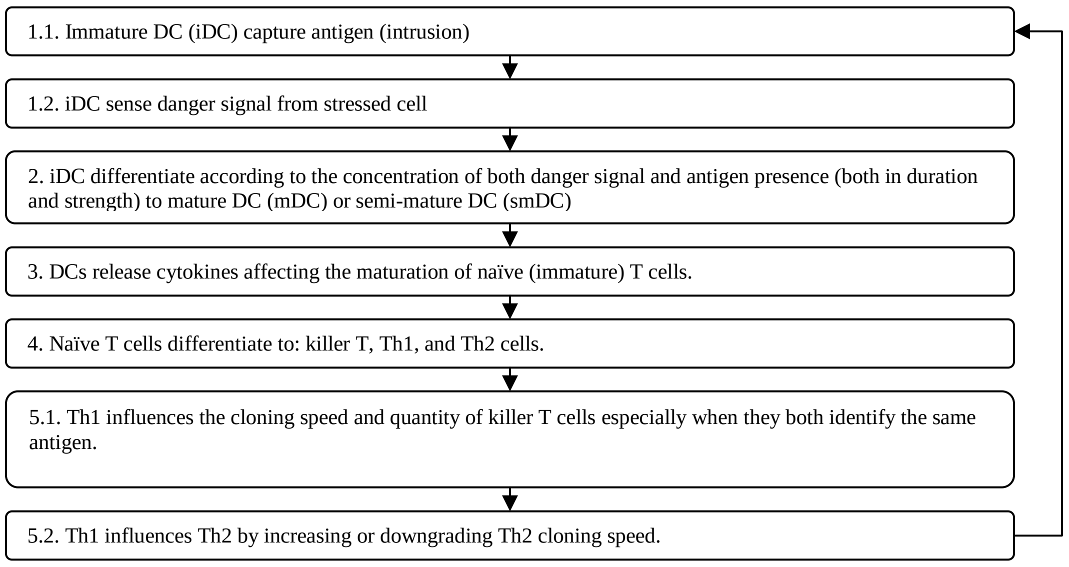


Figure 7.1. iDC, and T cell differentiation.



1.1. Immature DC (iDC) capture antigen (intrusion)

1.2. iDC sense danger signal from stressed cell

2. iDC differentiate according to the concentration of both danger signal and antigen presence (both in duration and strength) to mature DC (mDC) or semi-mature DC (smDC)

3. DCs release cytokines affecting the maturation of naïve (immature) T cells.

4. Naïve T cells differentiate to: killer T, Th1, and Th2 cells.

5.1. Th1 influences the cloning speed and quantity of killer T cells especially when they both identify the same antigen.

5.2. Th1 influences Th2 by increasing or downgrading Th2 cloning speed.

Figure 7.2. Flow chart of DC and T cell differentiation

7.3.1. Implementation

The iDC component of the danger theory is the controlling element of all subsequent activities of intrusion detection systems. iDC is responsible for sensing the system condition and indicates if the system is under attack or not by not only identifying the existence of intrusion instances but also by noticing danger conditions of the system resources. Therefore, iDC will react not only to deviations in the normal system call sequences but also will react if any monitored system resource condition exceeds a predefined acceptable threshold. The APC component (consisting of both iDC and DC) is implemented in our system as two classes: iDC class and DC class. If we consider APC as a black box then the input to this component is the following:

- PAMP: intrusion signatures (i.e. mismatches between current monitored system calls and pattern entries in the normal database).
- Danger signal: high CPU and memory usages.
- Safe signal: normal CPU and memory usages.
- IC: user is present or not.

The output of this component will be as follows:

- Mature Cytokines: intrusion detected.
- Semi-mature Cytokines: normal or acceptable behavior.

There are unlimited ways to calculate the output of APC. The security requirements of each system can control such process. However, in our implementation the following were considered while trying to indicate if the system is under attack or not:

- Current CPU usage.
- Previous CPU usage within a previously specified period.
- Current memory usage.
- Previous memory usage within a previously specified period.
- Current use of abnormal signals.
- Previous use of abnormal signals within a previously specified period.
- Current occurrence of a mismatch.
- Previous occurrences of mismatches within a previously specified period.

For each of the previous elements, an importance indicator is associated with it. This indicator affect the overall value of the safe or danger signal calculated when all of these elements are combined. For example, if the importance of the current CPU value is 10% when compared to the other elements and the importance of a current mismatch is 50 % when compared to the remaining elements, the occurrence of a mismatch will affect the overall indication of an intrusion more heavily than the occurrence of a high CPU burst. Such decision is heavily drawn from both the normal and intrusive behavior of a system. For example, if an intrusion is usually associated with high memory usage than with mismatch occurrence then memory usage is given a higher percentage when calculating the finial system condition value. However, in any case all elements in the list are included because they all have an affect on the finial decision but with different strengths.

For both CPU and memory usage values, the system keeps the following:

- MIN_ACCEPTABLE_THRESHOLD_VALUE
- MAX_ACCEPTABLE_THRESHOLD_VALUE.

If CPU's current value lies between both threshold values then it is normal and a normalized value is calculated depending on where it falls within this normal range.

For example suppose that the following are previously defined:

```
CPU_MIN_ACCEPTABLE_THRESHOLD_VALUE;
CPU_MAX_ACCEPTABLE_THRESHOLD_VALUE;
Then if (current_CPU_value <  CPU_MAX_ACCEPTABLE_THRESHOLD_VALUE)
Safe_normalized_CPU_value = ((current_CPU_value * 100) /
    CPU_MAX_ACCEPTABLE_THRESHOLD_VALUE) / 100;
```

This will help identifying where does this normal value of CPU fall in the range of acceptable behavior of the CPU. A similar procedure is used to calculate danger_normalized_CPU_value if the current CPU value read is above the max threshold value.

After performing the same procedure for all signals under consideration, current and previous values of these signals are used to calculate the normalized value of C_P, C_S and C_D (C_P: normalized value of all signals related to PAMP condition, C_S : normalized value of all signals related to safe condition, and C_D: normalized value of all signals related to danger condition). Equation 2 is used to evaluate whether the system in under attack by producing high mature DC values or in normal condition by producing high semi-mature DC values. It is important to mention that we have calculated the concentration of a signal with respect to the strength of an associated signal as well as duration of the steady signal. The weights in equation 2 are obtained from table 7.2.

$$C_{[csm,semi,mat]} = \frac{((W_P * C_P) + (W_S * C_S) + (W_D * C_D))}{(W_P + W_S + W_D)} * \frac{1 + IC}{2} \qquad (2)$$

W	**csm**	**semi**	**mat**
PAMP	2	0	2
Danger signal	1	0	1
Safe signal	2	3	-3
IC	1	1	1

Table 7.2. Weights of different danger theory parameters used to calculate the different cytokine concentrations of DC.

The intrusion detection system employing iDC and DC techniques is implemented as an object oriented program where each cell is represented as a class. The code of the program is available in Appendix F. In summary, the pseudo code of the activity diagram of iDC is shown in Figure 7.3. iDC cell (object) is responsible for monitoring the system calls generated by an application and as the sequence reaches a specific window size, it is compared to the entries in the associated database. If it does not match any entry, it is considered an anomaly. At the same time signals are gathered from the system. Mainly we are testing the memory and CPU usages. According to different inputs such as the current and previous conditions of several monitored activities, the concentration signals of safe, danger, PAMP and IC are calculated. These values are then sent to the DC object for further processing.

Figure 7.4. is the pseudo code of the activity diagram of DC. After receiving the concentration values of danger signal, safe signal, PAMP and IC from iDC object, it calculates the DC concentration value and decides if the system is in a dangerous or safe normal environment. If it is the result of a dangerous situation then mature DC will be high and DC will send a PRIME message to Th1 along with the string that was handled and what is the source of this danger. The danger source can be a result of a mismatch, or high CPU, high memory usage or a combination of any of them. If semi-mature DC was the outcome then DC will send a SUPRESS message to Th1.

```
0 Do
1       Read system call
2 Until string reached window size
3 Match string with entries in database
4 If (identified bacteria)
5 Then
6      Sense signal values
7      Calculate $C_{PAMP}$, $C_{safe}$, $C_{danger}$, IC
8      Send values to DC object
9 Else go to 0
```

Figure 7.3. Pseudo code of the activity diagram of iDC

```
0 Read $C_{PAMP}$ :normalized value of mismatch detection
1 Read $C_{safe}$ : normalized value of normal behavior
2 Read $C_{danger}$ :normalized value of dangerous behavior
3 Read IC: normalized value of presence of user
4 Read sources
5 Read string
6 calculate DC concentration cytokine values
7 if (semi-DC > mat-DC)
6      send Suppress message to Th1
7     send source value to Th1
8     Send string value to Th1
9 Else
10     send prime message to Th1
11     send source value to Th1
12     Send string value to Th1
13 end else
14 go to 0
```

Figure 7.4. Pseudo code of the activity diagram of DC

7.3.2. Performance

Figure 7.5. (a), (b), (c) are sample output of running the objects iDC and DC. In general, we are logging the subsequence that is currently being processed. In our example the window size is = 4 and if the tested subsequence does not exist in the normal database, a message is displayed indicating that it is a mismatch. We keep track of whether the user is present or not and we read the CPU and Memory usage values. Then, we check if one of the system calls read in the handled subsequence is a dangerous signal or not. With dangerous signal we mean that it is one of the signals that are abnormally used to kill a process intentionally. After reading these values the appropriate data structure holding the previous values of a parameter is updated. Then DC will use these current and previous values to calculate the DC cytokine value. It will be either semi-mature “semi” indicating that the system is still under safe acceptable condition or mature “mat” indicating an intrusion.

```
Handeled Window:    2   3   4   7
Is a mismatch
User present: 1
CPU usage: 30.8
Mem usage: 40.6
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    0   0   0   0   1   0   0   0   0   0
previous IC:    1   0   0   0   0   0   0   0   0   0
Semi


Handeled Window:    3   4   7   6
Is a mismatch
User present: 1
CPU usage: 30.8
Mem usage: 40.6
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    0   0   0   0   1   1   0   0   0   0
previous IC:    1   1   0   0   0   0   0   0   0   0
Semi


Handeled Window:    8   5   4   6
Is a mismatch
User present: 1
CPU usage: 30.8
Mem usage: 40.6
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   0   0   1   1   1   1   1   0
previous IC:    1   1   1   1   1   1   1   0   0   0
Semi
```

Figure 7.5. Sample output of running iDC and DC enhanced IDS. “Handled window”: string currently processed. “Is a mismatch”: does not exist in the database or “is normal”: exist in the database. “User present”: 1 is present, 0 is not. “CPU usage”: percentage of CPU usage. “Mem usage”: percentage of memory usage. “is an abnormal signal”: 0 → no, 1→ yes. “previous CPU”, “previous abnormal”, “previous mismatches”, “previous IC”: 0→low, 1: high. “Semi” or “mat”: indicate the resulting condition of the system: semi→ safe, mat→ dangerous.

Table 7.3., shows the performance parameters collected while running the iDC and DC enhanced IDS and compares it with the values obtained when running the original lookahead-pairs IDS. As observed both have exactly the same performance in terms of testing time and storage requirements.

Parameter	Lookahead pairs IDS	iDC and DC enhanced IDS
W	4 system calls	4 system calls
Testing time	0 seconds	0 seconds
# pairs in Normal DB	1029	1029
Space cost while running (bytes)	24576	24576
Space cost while saved to disk (bytes)	4116	4116
# pairs in testing	4044	4044

Table 7.3. Performance comparison between lookahead pairs IDS and lookahead-pairs method enhanced with iDC and DC IDS

In general, several types of attacks can exploit a system. An attack can result from accumulative mismatches, accumulative high CPU usages, accumulative high memory usages, accumulative or single use of abnormal signals or a combination of any of them.

Lookahead pairs based IDS will only be able to detect an attack that involves mismatches between the sequences currently under consideration and normal database. It will not be able to detect attacks that deviate producing mismatches or exceeding the allowable threshold value. However, the iDC and the DC enhanced IDS will be able to detect more attack types.

In general, if a system is fully trained, any identified mismatch must indicate an intrusion and such system will not produce any false positives. However, it is important to note that if the system is not fully trained, the lookahead-pairs method will give a false positive rate equal to the number of mismatches identified while testing that may include normal behavior. However, the iDC and DC enhanced IDSs will balance its decision not only on identifying mismatches but also the environment condition (i.e. signals collected at the time the mismatch was identified) and accordingly will decide if it is an intrusion or not.

In the case of false negatives, where the system misses an intrusion instance, lookahead pairs method performs poorly. The original lookahead-pairs method based IDS does not miss any mismatches but it misses intrusions that deviate producing a number of mismatches that exceed the specified threshold. As a result the iDC and DC enhanced IDS will identify the intrusions associated with high CPU and memory usages as well as using abnormal signals the lookahead pairs method will miss such attacks producing higher false negative rate.

7.4. Positive and Negative Detector Sets

Generating suitable patterns has been a hot topic for several years. Identifying patterns that can represent the considered problem is a very important issue especially for artificially immune system based IDS. Building a database with positive detectors means that the database contains instances of normal behavior. However, a database with negative detectors means that the database has instances of the complement of the normal behavior. The advantages and disadvantages of each type can be found in [Esponda, Forrest and Helman 2004]. In general, for a small sized problem, positive detector sets outperform negative detector sets especially with regard to storage requirements. What we mean with small sized problem is that the normal behavior can be identified in a limited and small number of sequences. The human immune body uses both negative and positive selection to perform different immune based functionalities. We found it important to implement both aspects and to compare between them. The system has two databases that are used to check for anomalies. One is used by the B cell and the other by iDC cell. If we choose to use either positive, negative or both selection algorithms, different scenarios can exist. For example, we may choose to have the B cell's database created with positive detectors or negative detectors sets and the same applies for the other database. In the following sections a comparison is presented.

7.4.1. Implementation

Simply the difference between positive and negative detector generation can be explained by the following example. If the number of system calls = 100 and are represented as {1, 2... 100}. If we choose a window size = 4 and have the following training sequences: { 1,2,3,4,5,6,7,8}.

The generated positive detectors will be:

{{1, 2, 3, 4}

{2, 3, 4, 5}

{3, 4, 5, 6}

{4, 5, 6, 7}

{5, 6, 7, 8}}

However, to generate negative detectors for this specific training sequence we need to find the complement sequences of the patterns found in the positive detector database. The number of entries in such a database if we are looking for a complete database is indefinite. For example, we will start doing the following:

{{1, 2, 3, and 5},

{1, 2, 3, 6},

{1,   2,   3,   7},

{1,   2,   3,   8},

 ….   }

As we attempt to fill the database, we find that the number of entries increases indefinitely. In our system we compared between using positive and negative detector generation with lookahead-pairs method. In our positive detector generation, we scan normal training files and then depending on the window size, we start creating the patterns. For each pattern, we generate the associated pairs and then set their respective locations in the appropriate set array to one. During testing we compare the currently tested sequence against the entries in the normal database and if it is not found then an anomaly is flagged.

Generating negative detectors can belong to two types: 1) negative detectors generated for methods that store patterns in a linked list data structure or the structure, or 2) negative detectors generated for methods that store patterns in an array such as lookahead-pairs method.

For the negative detectors generated for methods that store patterns in a linked list or a tree data structure such as sequence method we perform the following steps. First, the normal behavior patterns are identified. Second, we randomly generate a candidate pattern of the same size and compare it with the patterns in the normal database. If the candidate pattern match any entry in the normal database, this candidate pattern is deleted and a new pattern is generated. Otherwise, the candidate pattern is added to the selected patterns list. We repeat this step until we have enough (i.e. previously identified) number of patterns. In the detection stage, the tested sequence is checked against the database and if a match occurs then an intrusion have been identified. Furthermore, as the number of patterns generated by negative selection increase, the storage cost increases.

For the negative detectors generated for methods that store pattern in an array such as lookahead pairs, we simply generate all possible normal pattern sequences. We then find the associated pairs and set their locations to ones. The database that is based on negative selection is the complement of the normal database. Meaning we complement the cell values of the arrays. If it is set to 1, we unset it and if it is not set, we set it. In this method we don’t need to randomly generate any sequence and no additional storage requirements are needed. Our code to carry out the comparison between positive and negative detectors can be found in Appendix E.

7.4.2. Performance

With a size-limited representation of normal behaviour, positive detectors are the better choice since the pattern database generated will be of acceptable size. This is true for methods that do not use the lookahead-pairs method data structure to store its database. With lookahead pairs method, both positive and negative detectors will occupy the same storage requirements and only requires the appropriate matching to be performed.

However, this is not the case with other methods that employ for example trees or linked lists to store their patterns. In such methods and if the normal behaviour is represented in a small finite manner, negative detectors will be infinite. In such a case we will be required to decide on the acceptable or allowable number of negative detectors that will represent our problem efficiently. The more allowable number of negative detectors the better the detection rate. As shown in table 7.4., we have tried to generate a pool of negative detectors while increasing the number of generated sequences to achieve a similar mismatch of that of positive detectors. It has been found that we are powered by the random generator that produce candidate detectors and may or may not cover the required set of patterns. For example, from table 7.4., 199 positive detector sequences can identify 67 mismatches. We would like to have a pool of negative detectors that are not only randomly generated but also identify almost 67 matches of anomalous behaviour. In particular the following has been noticed from table 7.4.:

- There is no guarantee that the negative detectors created will cover better the intrusive instances even if the allowable number of detector sequences is increased.
- We need to correctly identify what should be randomly generated. From table 7.4., sequences of the complement of normal behaviour have been randomly generated. If we decide to use these complement sequences with lookahead-pairs method, high false positives will result such as in the case of having 95 matches with 900 detector sequences. This can be explained by the following example:

Suppose we have the following normal sequence {1, 2, 3, 4}, the pair (2, 1) will be added to the normal positive selection database at set 1. If we are generating the complement of the normal sequence, the random generator will provide the sequence {1, 2, 3, and 5} as an acceptable pattern. If we convert this sequence to its lookahead-pairs equivalent, the pair (2, 1) will be added to the negative selection database at set 1 as well. Such pairs will be responsible for producing false positives.

	Pos.	Neg.														
# mismatches	67															
# matches		30	36	6	8	10	11	20	3	22	61	79	16	69	95	40
# pairs	500	537	539	540	596	595	597	898	896	898	2073	2073	2075	2670	2662	2669
# detector sequences	199	180	180	180	199	199	199	300	300	300	700	700	700	900	900	900
Cost saved to disk	6000	6444	6468	6480	7152	7140	7164	10776	10752	10776	24876	24876	24900	32040	31944	32028

Table 7.4. Performance comparison among positive and several negative pattern generation.

7.5. Danger Theory

Matzinger [Matzinger 1994] introduced the idea that the immune system does not respond only to foreignness depending on self-non-self discrimination but also to danger signals generated when a cell dies an unnatural way. Many cells such as B cell, Th1 cell, Th2 cell, APC, and killer T cell play a role in the adaptive reaction of the immune system and especially with danger theory. B cell is responsible for identifying bacteria (intrusion instances) and presenting it to Th1. APC is responsible for processing the environment and identifying bacteria as well as distress signals and presenting it to Th1. Th1, Th2 and Killer T cells are responsible for managing the immune reaction.

Previous attempts to model immune systems were implemented as a pool of agents mimicking different cells. In our implementation we are instantiating only one instance or object of each cell type. For example, rather than having a pool of B cells, and each cell identifies a number of antigens or intrusions, we have implemented an object B cell that is associated with a database of all possible patterns and it reads a sequence and compares it with this database. The same is applied to all other cells.

7.5.1. Implementation

Our implementation of danger theory concepts are summarized in the object diagram displayed in Figure 7.6. Figure 7.7. shows the associated data flow diagram of the activities performed by the different cells (objects).

B cell object is responsible for capturing antigens or intrusion instances. This is achieved by comparing a suitable sized sequence with the entries in the database of normal behavior associated with B cell object.

The iDC object is responsible for identifying intrusion instances as well as gathering statistics from the system such as memory and CPU usages. The collected data by the iDC object is then translated to concentrations of danger, safe, IC and PAMP signals and then they are passed to the DC object.

The DC object then decides if this information corresponds to an intrusion or if it is normal. Both B cell and DC cell objects present their information to the Th1 cell object that is responsible for priming Th2 and killer T cells in case of danger and suppressing Th2 in case of safe operation.

Th2 then either suppresses or primes B cell object. In case of intrusion, killer T cell will display a message to the user with the type of intrusion identified. This will include the sources of such intrusion. For example, it will indicate if it is a mismatch, high memory, high CPU usages, or a combination of them. It will also display other related information such as what is the system calls that caused the mismatch and what application.

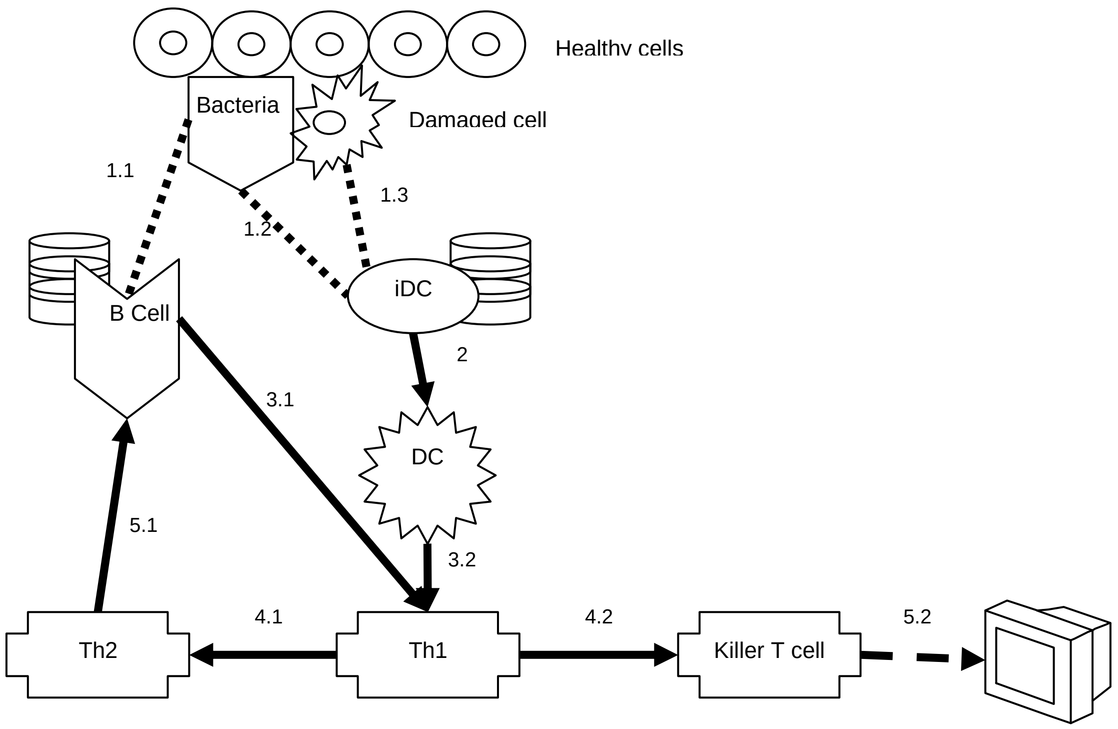
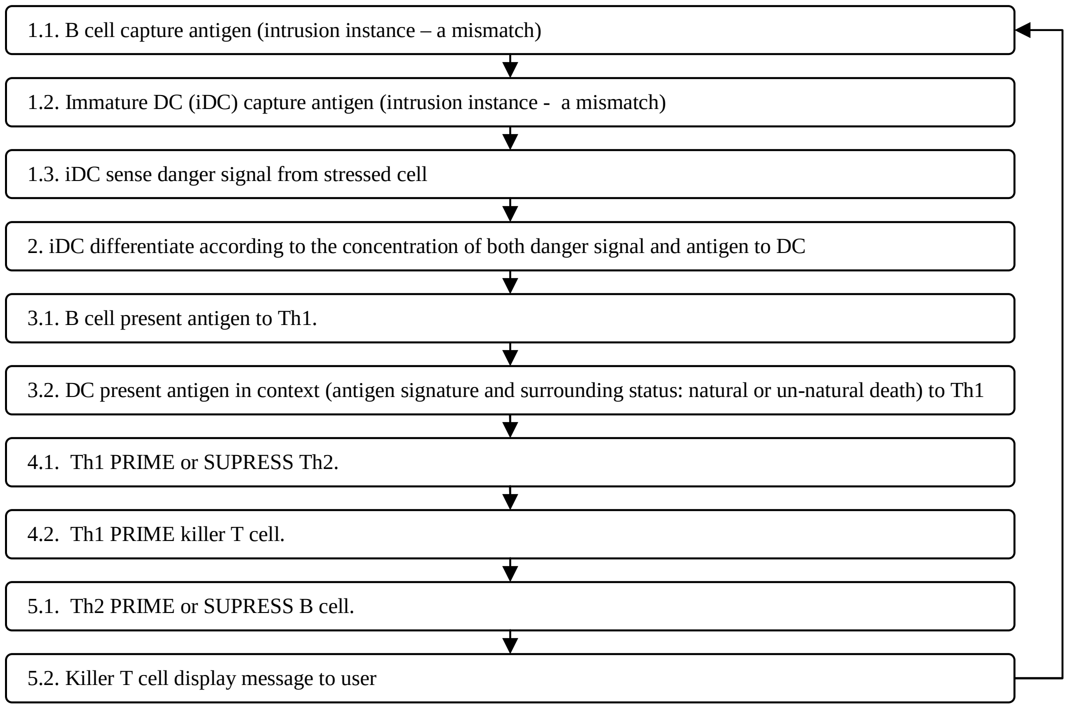


Figure 7.6. Danger theory system overview architecture.



1.1. B cell capture antigen (intrusion instance – a mismatch)

1.2. Immature DC (iDC) capture antigen (intrusion instance - a mismatch)

1.3. iDC sense danger signal from stressed cell

2. iDC differentiate according to the concentration of both danger signal and antigen to DC

3.1. B cell present antigen to Th1.

3.2. DC present antigen in context (antigen signature and surrounding status: natural or un-natural death) to Th1

4.1. Th1 PRIME or SUPRESS Th2.

4.2. Th1 PRIME killer T cell.

5.1. Th2 PRIME or SUPRESS B cell.

5.2. Killer T cell display message to user

Figure 7.7. flow chart of the steps carried out by the artificial immune based IDS employing danger theory concepts

Implementing this system required the definition of several classes to handle the different functionalities of the danger theory based IDS that are:

- iDC
- DC
- B cell
- Th1
- Th2
- Killer T
- Exit engine
- IDS system

The code of the different objects can be found in Appendix F. Both iDC and DC activity has been explained in section 6.3 of this dissertation. The B cell object is responsible for reading system calls produced by the application and then comparing subsequences with entries in the database. The database in our system consists of normal behavior patterns. If the tested subsequence does not match any entry in the database then an intrusion is flagged. The B cell object then informs Th1 object of the mismatch and which subsequence caused it. At the same time, the B cell object listens to Th2 which can send a suppress or a prime signal to B cell object. Th2 informs B cell object of the string under consideration and whether it is an intrusion by the prime signal or a normal activity by the suppress signal. For each entry in the B cell database there is an associated threshold value. This threshold value indicates how many times an entry has been seen or has matched a sequence. A minimum acceptable threshold as well as a maximum acceptable threshold values are previously identified by an officer and indicate the minimum and maximum values that govern the lifespan of an entry in the B cell database. If Th2 primes B cell then the threshold value associated with the string is increased. If Th2 suppresses B cell then the associated threshold value is decreased. If the threshold goes below the minimum accepted threshold then this entry is removed from the database. We decided to perform such action especially because the system is usually not fully trained and some new sequences will cause a mismatch but will not be associated with any dangerous signals. Therefore, our system should tolerate such new changes. The pseudo code of the activity diagram of B cell is shown in Figure 7.8. When there is no more system calls to be examined or when the IDS is shut down, B cell object calls ExitEngine object which is responsible for finalizing IDS activities.

Th1 object can receive input from both B cell object and DC objects. If Th1 receives input from B cell and DC object, then it will check the threshold value associated with the string sent by B cell object. If it exceeds the maximum acceptable threshold value no confirmation from DC is required to prime killer T cell and an intrusion is flagged. If the threshold value is within an acceptable range but DC sent a prime message then killer T cell is primed. In both cases Th2 is primed. Otherwise a suppress message is sent to Th2. If Th1 receives input from B cell only, the threshold value is checked. If it exceeds the threshold value then Th2 and killer T cell objects are primed; otherwise Th2 is suppressed. If Th1 receives input from DC only, it checks its cytokine value. If it is a mature signal, then killer T cell is primed; otherwise nothing is activated. The pseudo code of the activity diagram of Th1 cell is shown in Figure 7.9.

```
Do
    if (receive input from Th2)
    then
        if (input = Suppress)
        Then
            decrement threshold-array of string
            if (threshold < MIN-Threshold)
            then remove entry from table
            end then
            end if
        end then
        else // input is Prime
            increment threshold-array of string
        end else
        end if
    end then
    end if
    Do
        Read system call
    Until string reached window size
    Match string with entries in database
    If (identified bacteria)
    Then
        send string to Th1
        send threshold-array to Th1
    end then
Until no more system calls to read
call ExitEngine
```

Figure 7.8. Pseudo code of the activity diagram of B cell

Th2 object is responsible for receiving suppress or prime messages from Th1 object and then contacting B cell object to either suppress or prime its actions regarding a specific sequence. The pseudo code of the activity of Th2 is shown in Figure 7.10.

Killer T cells are responsible for responding to any attack against the immune human system by attacking the invader cells. In computer security killer T cell response can be by: 1) displaying a detailed message to the security officer explaining the intrusion conditions, 2) killing the process responsible for the attack, or 3) slowing down carrying out the system call which in many cases can defeat an attack. In our implementation and because we are performing off-line analysis, we choose the first option which is displaying the conditions of the attack. Therefore, killer T cell object displays to the user or security officer the type of attack and conditions resulting from such attack. The pseudo code of the activities of killer T cell is shown in Figure 7.11.

```
0 if (received input from B and DC)
1 then
0         if (threshold of string > MAX-allowable-threshold)
0         Then
0                 send Prime message to killer T
0                 send prime message to Th2
0         end then
0         end if
0         if (mature DC)
0         then
0                 send prime message to killer T
0                 send prime message to Th2
0         end then
0         else
0                 send suppress message to Th2
0         end else
0         end if
0 end then
0 else    if (received input from B and not from DC)
0         then
0                 if (threshold of string > MAX-allowable-threshold)
0                 Then
0                         send Prime message to killer T
0                         send prime message to Th2
0                 end then
0                 else
0                         send suppress message to Th2
0                 end else
0                 end if
0         end then
0         else    if (received input from DC and not from B)
0                 then
0                         if (mature DC)
0                         then
0                                 send prime message to killer T
0                         end then
0                         end if
0                 end then
0                 end if
0         end else
0         end if
0 end else
0 end if
```

Figure 7.9 Pseudo code of the activity diagram of Th1

```
0 Receive string and attack type from Th1
1 if (attack type = suppress)
2 then
3       send suppress message to B
4       send string to B
5 else
6      send prime string message to B
8 end else
9 go to 0
```

Figure 7.10. Pseudo code of the activity diagram of Th2

```
0 Receive string from Th1
1 Receive source from Th1
2 Receive system identifying engines from Th1
3 print string to user
4 print source to user
5 print identifying engines to user
```

Figure 7.11. Pseudo code of the activity diagram of Killer T cell

```
0   Receive exiting IDS message
1   if exit message from B cell
2   then
3         if received exit message from iDC
4         then
5               Finalize exiting IDS program
6               Print statistics
7               Goto 18
8        Else go to 0
9   else if exit message from iDC
10  then
11        if received exit message from B cell
12        then
13              Finalize exiting IDS program
14              Print statistics
15              Goto 18
16       Else go to 0
17 End then
18 Exit program
```

Figure 7.12. Pseudo code of the activity diagram of Exit Engine

ExitEngine is an object that is responsible for making sure that the IDS is shut down and the statistics are displayed only when both B cell and iDC cells have existed. The pseudo code of ExitEngine object is shown in Figure 7.12.

7.5.2. Performance Comparison

In this section we will compare the performance of the original lookahead pairs method IDS with other enhanced versions that use the same data structure used by lookahead pairs method. Enhancing danger theory method was performed in 3 steps:

1. iDC and DC functionalities of danger theory were added to the original lookahead pairs method. This means that we are not only looking at mismatches but also searching for distress signals associated with such mismatches.
2. Added the functionality of B cells with it associated database. In this case we have two databases to compare against. One associated with B cells and another with iDC.
3. All cell components of danger theory are added and examined.

In general, a danger theory based IDS can identify the following:

- Mismatch identified by B cell only.
- Mismatch and distress identified by DC cell only.
- Mismatch by B and DC cells and distress by DC cells.

Whereas the original lookahead-pairs method IDS can only identify intrusions caused by a mismatch.

In Table 7.5., we have identified 9 scenarios and compared the performance of the four IDSs. The 9 scenarios explain the different attack types a system may encounter. The different scenarios are related to the different controlling factors that indicate an intrusion such as the existence of a mismatch, high CPU usage and high memory usage. We have compared four systems. The original lookahead-pairs method IDS can only identify intrusions producing a number of mismatches that exceed a previously identified threshold value. It will not identify an attack that deviate or produce mismatches that don't exceed the threshold value but produce dangerous signals. The dangerous signals checked in our system are the high CPU and high memory usages. False positives can also be encountered with lookahead-pairs method especially when there is a mismatch identified but with no associated dangerous signal. This could happen when the system is not fully trained on all possible system call sequences and a new sequence may be produced. Lookahead pairs will identify it as an intrusion despite being sometimes normal. When enhancing lookahead-pairs method with the functionality of iDC and DC, all attacks will be identified. If we are limited with space requirements, enhancing an IDS with iDC and DC characteristics is a good option. Because such enhancement will not have any significant additional storage and processing time affects. However, it will enhance detection tremendously and lower false positives.

Incorporating all cell types of danger theory especially B and iDC who are associated with a database has similar detection rate obtained from the iDC enhanced IDS. Meaning that it does not identify any additional types of attacks. However, it makes the system more robust because if one of the databases gets tampered with the other database will be able to identify the attack. This, of course comes with a cost of doubling the storage requirements of the IDS. The databases of B and iDC cells can hold either positive or negative detector patterns. In our 4 systems we are using positive detector database. However, since lookahead-pairs is not affected with the size of complement sequences of normal behavior, both databases will have the same storage requirements. One of the benefits of using negative detectors is the ability to distribute the system to other systems easily.

Table 7.6. compares the performance of the 4 versions of lookahead-pairs IDS and its enhanced version with respect to storage requirements. Both the original version of lookahead pairs and its enhanced version with iDC and DC have the same storage requirements. However, if we use another database for B cell the storage requirement is doubled.

CPU attack	**MEM attack**	**Mismatch attack**	**Lookahead pairs**	**Lookahead pairs & iDC**	**Lookahead pairs & danger theory with +ve detectors**	**Lookahead pairs & danger theory with +ve & -ve detectors**
0	0	0	Y	Y	Y	Y
0	0	1	Y	Y	Y	Y
0	1	0	N	Y	Y	Y
0	1	1	Y	Y	Y	Y
1	0	0	N	Y	Y	Y
1	0	1	Y	Y	Y	Y
1	1	0	N	Y	Y	Y
1	1	1	Y	Y	Y	Y
Added characteristic				Better detection	Robust	Distribution

Table 7.5. Intrusion types identified by the four types of IDS: lookahead pairs IDS, iDC enhanced, danger theory enhanced with positive detectors for both B and iDC, and danger theory enhanced with positive and negative detectors for B and iDC. 0: No attack. 1: an attack. Y: yes. N: no. FP: result in false positives.

	Lookahead pairs	Lookahead pairs & iDC	Lookahead pairs & danger theory with +ve detectors	Lookahead pairs & danger theory with +ve & -ve detectors
Total testing time (sec)	0	0	0	0
Space cost of 1 DB while running	65536	65536	65536	65536
Space cost of 2 DBs while running	65536	65536	131072	131072
Space cost of 1 DB while saved to disk	10896	10896	10896	10896
Space cost of 2 DBs while saved to disk	10896	10896	21792	21792+

Table 7.6. Performance comparison of 4 versions of lookahead-pairs and its enhanced systems.

7.6. Validation and Verification

One technique used to validate and verify our modified intrusion detection system is the conference test as a dynamic functional testing. Dynamic testing involves executing the system by choosing a number of test cases and its input test data. The input test cases are used to determine output test results. With functional testing, we identify and test all functions of the system as defined in requirements. With the conference test, we choose different input values then design test cases that invoke every functional requirement in the specification at least once. Our validation hypothesis is that the danger theory based intrusion detection system will only pass the conference test if an only if there is no failures. The purpose of validating the implemented system is governed with the purpose of the new enhanced system. The purpose of this project is to enhance the lookahead-pairs method based intrusion detection system with danger theory concepts and enhance the detection rate of the original system. The purpose of the validation phase is to make sure that given different input sets that the output produced by the system is correct. In general, the intrusion detection system takes as an input the following:

- System-call sequences that contain both normal and intrusive instances.
- CPU current values
- Memory usage current values

System calls are read one by one and then are formatted as a sequence. The appropriate number of system calls is grouped to form a testing subsequence; they are checked against the database. As long as no intrusive instance is discovered the system continues monitoring system calls. In case an intrusion has been discovered the values for the CPU and memory usages are read. At the same time, on equal intervals the CPU and memory usage values are read to check for attacks that do not involve mismatched system calls. Table 7.7. shows the different test cases used to test the system output and compare between the original and enhanced systems outputs. In total there are 64 test cases. We have 6 different input combinations that are the current values of CPU usage, memory usage and identification of a mismatch. We are also examining the previous immediate condition of the system. The “expected output” column indicates what the system should produce given the specified conditions. The “actual output” is the output produced by the system. When comparing both the lookahead-pairs method with the enhanced version with danger theory we calculated the number of accurate outputs produced. Both the original lookahead pairs method IDS and the enhanced system were able to identify correctly all 64 case inputs. Accuracy here does not mean that the system identified an intrusion correctly but for validation and verification purposes, it means that the system performed what it is supposed to perform correctly. Figure 7.13. is an example of the output produced by the enhanced IDS when testing against test case 8. In this case a current mismatch is identified and is associated with high CPU and memory usages. Appendix H is an example output produced when testing the enhanced system with test case 10. In this test case we are making sure that the enhanced IDS identify a contiguous mismatch attack. Although the lookahead-pairs method enhanced with danger theory IDS is best known to identify attacks with danger signals, the system should be able to identify as well attacks that do not cause dangerous signals. A mismatch threshold is identified for the system and if a number of mismatches exceed this number the system will display an attack message to the user.

Test Case	**Previous (N) High CPU values**	**Previous (N) High MEM values**	**Previous (N) identified mismatches**	**Current High CPU**	**Current High MEM**	**Current Identified Mismatch**	**Desirable output**	**Lookahead pairs with mismatch threshold = N**		**Lookahead-pairs method enhanced with danger theory**	
								Expected output	Actual output	Expected output	Actual output
1	0	0	0	0	0	0	normal	normal	normal	normal	normal
2	0	0	0	0	0	1	normal	normal	normal	normal	normal
3	0	0	0	0	1	0	normal	normal	normal	normal	normal
4	0	0	0	0	1	1	attack	normal	normal	attack	attack
5	0	0	0	1	0	0	normal	normal	normal	normal	normal
6	0	0	0	1	0	1	attack	normal	normal	attack	attack
7	0	0	0	1	1	0	normal	normal	normal	normal	normal
8	0	0	0	1	1	1	attack	normal	normal	attack	attack
9	0	0	1	0	0	0	attack	attack	attack	attack	attack
10	0	0	1	0	0	1	attack	attack	attack	attack	attack
11	0	0	1	0	1	0	attack	attack	attack	attack	attack
12	0	0	1	0	1	1	attack	attack	attack	attack	attack

13	0	0	1	1	0	0	attack	attack	attack	attack	attack
14	0	0	1	1	0	1	attack	attack	attack	attack	attack
15	0	0	1	1	1	0	attack	attack	attack	attack	attack
16	0	0	1	1	1	1	attack	attack	attack	attack	attack
17	0	1	0	0	0	0	attack	normal	normal	attack	attack
18	0	1	0	0	0	1	attack	normal	normal	attack	attack
19	0	1	0	0	1	0	attack	normal	normal	attack	attack
20	0	1	0	0	1	1	attack	normal	normal	attack	attack
21	0	1	0	1	0	0	attack	normal	normal	attack	attack
22	0	1	0	1	0	1	attack	normal	normal	attack	attack
23	0	1	0	1	1	0	attack	normal	normal	attack	attack
24	0	1	0	1	1	1	attack	normal	normal	attack	attack
25	0	1	1	0	0	0	attack	attack	attack	attack	attack
26	0	1	1	0	0	1	attack	attack	attack	attack	attack
27	0	1	1	0	1	0	attack	attack	attack	attack	attack
28	0	1	1	0	1	1	attack	attack	attack	attack	attack
29	0	1	1	1	0	0	attack	attack	attack	attack	attack
30	0	1	1	1	0	1	attack	attack	attack	attack	attack
31	0	1	1	1	1	0	attack	attack	attack	attack	attack
32	0	1	1	1	1	1	attack	attack	attack	attack	attack
33	1	0	0	0	0	0	attack	normal	normal	attack	attack
34	1	0	0	0	0	1	attack	normal	normal	attack	attack
35	1	0	0	0	1	0	attack	normal	normal	attack	attack
36	1	0	0	0	1	1	attack	normal	normal	attack	attack
37	1	0	0	1	0	0	attack	normal	normal	attack	attack
38	1	0	0	1	0	1	attack	normal	normal	attack	attack
39	1	0	0	1	1	0	attack	normal	normal	attack	attack
40	1	0	0	1	1	1	attack	normal	normal	attack	attack
41	1	0	1	0	0	0	attack	attack	attack	attack	attack
42	1	0	1	0	0	1	attack	attack	attack	attack	attack
43	1	0	1	0	1	0	attack	attack	attack	attack	attack
44	1	0	1	0	1	1	attack	attack	attack	attack	attack
45	1	0	1	1	0	0	attack	attack	attack	attack	attack
46	1	0	1	1	0	1	attack	attack	attack	attack	attack

47	1	0	1	1	1	0	attack	attack	attack	attack	attack
48	1	0	1	1	1	1	attack	attack	attack	attack	attack
49	1	1	0	0	0	0	attack	normal	normal	attack	attack
50	1	1	0	0	0	1	attack	normal	normal	attack	attack
51	1	1	0	0	1	0	attack	normal	normal	attack	attack
52	1	1	0	0	1	1	attack	normal	normal	attack	attack
53	1	1	0	1	0	0	attack	normal	normal	attack	attack
54	1	1	0	1	0	1	attack	normal	normal	attack	attack
55	1	1	0	1	1	0	attack	normal	normal	attack	attack
56	1	1	0	1	1	1	attack	normal	normal	attack	attack
57	1	1	1	0	0	0	attack	attack	attack	attack	attack
58	1	1	1	0	0	1	attack	attack	attack	attack	attack
59	1	1	1	0	1	0	attack	attack	attack	attack	attack
60	1	1	1	0	1	1	attack	attack	attack	attack	attack
61	1	1	1	1	0	0	attack	attack	attack	attack	attack
62	1	1	1	1	0	1	attack	attack	attack	attack	attack
63	1	1	1	1	1	0	attack	attack	attack	attack	attack
64	1	1	1	1	1	1	attack	attack	attack	attack	attack
Accuracy								100%		100%	

Table 7.7. Test cases used to validate the lookahead pairs method enhanced with danger theory.

```
current testing input row:  90 125 106 5
mismatch pair: <5,106> at plain: 0
mismatch pair: <5,125> at plain: 1
mismatch pair: <5,90> at plain: 2
Handeled Window:   90 125 106   5
Is a mismatch
User present: 1
CPU usage: 97.3
Mem usage: 97.4
Is an abnormal signal: 0
previous CPU:    1   0   0   0   0   0   0   0   0   0
previous memory:    1   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   0   0   0   0   0   0   0   0   0
previous IC:    1   0   0   0   0   0   0   0   0   0
Mat
```

Figure 7.13. Sample output of the lookahead-pairs method enhanced with danger theory IDS of test case 8. A mismatch has been identified and high CPU and memory usages have been noticed. The output is "Mat" indicating mature DC or danger.

In table 7.7, we verity that the system is implemented correctly and also helped us to validate or demonstrate that the results obtained are correct, which is an example of trace validation.

In general, to validate the enhanced system requires a two fold validation. Since the enhanced system is based on an already built and validated system the first part of validation is checked. The original lookahead pairs method has been validated in section 5.8. of this dissertation.

The added functionalities of the danger theory to the original lookahead pairs method will be validated next. Since the enhanced system is composed of the original lookahead pairs method to create its normal database, the system or in particular the generated datasets are not sensitive to a particular input dataset. The other part of the enhanced system which is testing a sequence and indicating if it results in an intrusion or not, has been validated as follows.

The same data set used to test the original lookahead pairs method has been tested with the enhanced system and both produced the same results and identified the same intrusions. The enhanced system is created to detect more intrusions instances that have been missed by the lookahead pairs method. Table 7.7. shows the output of running the original and enhanced system over a set of test cases and shows that the enhanced system identified more attacks.

From table 7.8., and when comparing the desirable output with the actual output produced by the original system and the enhanced system, we note that the original system had a 57.8% detection rate and the enhanced system identified all intrusions.

	Original lookahead pairs method IDS	Lookahead pairs method enhanced with danger theory IDS
Number of attacks identified out of 64	37	64
Accuracy or detection rate = number of identified attacks / number of expected attacks to be identified.	57.8%	100%

Table 7.8. Detection rate comparison.

7.7. System Evaluation

An intrusion detection system is evaluated according to their performance in the following areas (evaluation criteria):

- Detection rate.
- False positive rate.
- False negative rate.
- Size of normal repository or the number of patterns in the normal database and their storage requirements.
- Speed of detection.

Our aim is building a system that will provide the following:

- High detection rate or in particular we aim of building a system that will detect more intrusion types and preferably novel attacks.
- Low false positive rate by not identifying normal behavior as an intrusion.
- Low false negative rate by not missing an intrusion.
- Small storage cost which results from smaller number of patterns and smaller pattern size.
- Fast detection speed.

In this dissertation we aimed to improve the performance of the original lookahead-pairs method IDS by incorporating danger theory's signal processing capability. In this section we will compare between the original lookahead-pairs method IDS and it's enhanced with danger theory IDS version with regard to the previously identified evaluation criteria. Table 7.9. elaborates on the evaluation of the criterions of the implemented IDSs.

Evaluation criteria	**Procedure used for evaluation**	**Observations**
Detection rate.	This is indicated in two folds: 1. The number of attacks identified from the total test cases tested in the system. 2. Mismatch threshold which is equal to the total number of pair-mismatches identified by the system divided by the number of pairs in the testing file.	Tables 7.7. and 7.8. indicate the number of attacks identified by the original lookahead-pairs method compared with the enhanced lookahead pairs method with danger theory. For the 64 attack scenarios, the enhance system was able to detect all attacks. However, the original system only had a 57.8% accuracy rate. We are performing a controlled experiment where we, ourselves, inject the attacks and check if they are identified or not. Both systems identified the same number of mismatches since this depends on the system calls that deviate from the patterns in the normal database.
False positive rate.	In a fully trained system the false positive rate is equal to 0.0, since any identified mismatch must be considered as an intrusion. However, usually, training the IDS on all possible patterns is not feasible. Different systems use different techniques to	With the original lookahead pairs method IDS, we simulated a false positive as mismatch that is not associated with any danger signal (high CPU or

	handle false positives such as: • Set an allowable mismatch threshold. If the number of mismatches identified exceeds this threshold then an intrusion is identified. However, it is difficult sometimes to identify a universal threshold value for patterns or sequences. • Ask the security administrator to manually indicate if this sequence is normal or intrusive. • Danger theory allows the system to experience mismatches and if they are moderate in number (i.e. don’t exceed a maximum threshold value) and are not associated with danger signals, to be added to the normal database. Our system has a minimum and maximum threshold values associated with mismatches. We have several scenarios: • If the mismatches exceed a maximum threshold and are not associated with dangerous signals, it is considered an intrusion and the information related to such instance is displayed to the user. If he decides to accept it, he must modify the database to include such pattern. • If the mismatches exceed the maximum threshold and is associated with dangerous signals then this is defiantly an intrusion. • If the number of mismatches is below the minimum threshold and is not associated with danger signals then it is considered normal. • If the number of mismatches is between the minimum and maximum allowable threshold and not associated with danger signals nothing is reported. • If the number of mismatches is between the minimum and maximum	memory usages). When the system is not fully trained, some new sequences will appear and will result in a mismatch since they do not exist in the normal database.

	allowable threshold value and danger signals are associated with it, the system uses an estimation equation (i.e. each parameter affecting the overall identification of an intrusion is given an associated percentage) to calculate whether there is an intrusion attempt or not. In general, our system removes any entry from the database that is identified later on as an intrusion instance. This is because our normal database is build using positive selection algorithm.	
False negative rate.	Many intrusions attempt to deviate the implemented IDS. The original lookahead-pairs method can not detect, for example, any pattern that is repeated indefinitely and exist in the normal database. The enhanced lookahead-pairs method with danger theory overcomes this shortcoming by monitoring other parameters as well as mismatches for intrusion instances. Our systems monitor for mismatches as well as CPU and memory usages associated with running processing. The CPU and memory usage parameters can be exchanged easily with any other appropriate parameter as seen fit for the security problem at hand. If an intrusion deviates the mismatch detection scheme which results in a false negative, the other parameters should (if chosen correctly) should indicate an intrusion instance.	With the original lookahead pairs method the false negative result from an intrusion that does not produce any mismatches. Such an intrusion is when the attacker, for example, writes an attack that will produce an acceptable sequence of system calls or a sequence that will produce very low mismatches that will not exceed the allowable mismatch threshold. Attacks that deviate the maximum threshold value but introduce overhead on other system parameters such as CPU or memory will be detected by the lookahead-pairs method enhanced with danger theory. The system enhanced with danger theory will detect both intrusions producing mismatches or inducing overhead on the CPU or memory of the system.

		However, any attack that does not affect these three parameters will not be detected.
Size of normal repository	For a given training data set and if the number of patterns generated for a specific window size w is N. If redundant entries are not removed then as the window size w increases by one the number of patterns N decreases by one. As shown in figure 7.14., the total number of patterns decreases by one as the window size increases. The number of patterns when the redundant entries are removed increase as the window size increases. The final number of patterns depends on the size of the log file used to train the system. It also depends on the frequency of a pattern appearing in the log file.	The more patterns added to the normal database the better the detection and lower false positives are generated. This is true with the original lookahead pairs method IDS. With the lookahead pairs method enhanced with danger theory we assumed that adding the signal processing functionalities will allow us to lower the number of patterns added to the database. This is true with sequence and variable length with overlap relationship methods because the size of the database is dependent on the number of patterns stored. With lookahead pairs method storage technique, to reduce memory requirements we need to remove one array or more from consideration.
Speed of detection.	This measured the time it takes for the off-line testing file to be read and each entry in it compared to the normal database entries and reporting the mismatches to the user.	Both lookahead-pairs method and enhanced lookahead-pairs method with dander theory IDSs finished processing and identifying the testing files (for the login and ps applications) in less than 1 second.

Table 7.9. IDS evaluation criteria

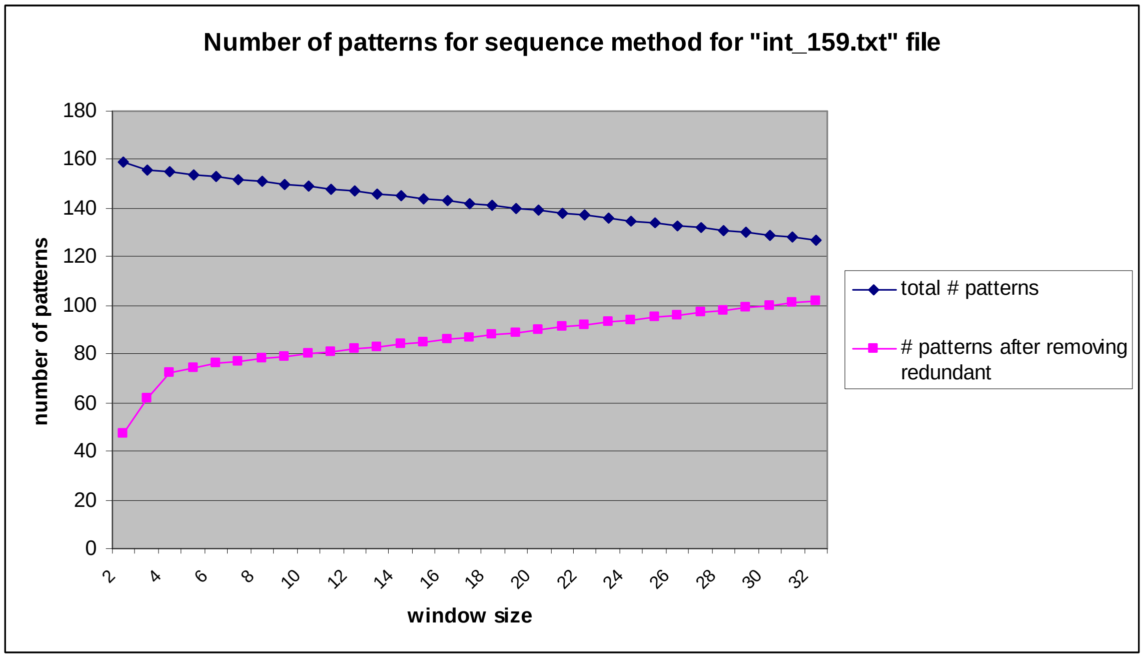


Figure 7.14. Number of patterns stored in the normal database if no redundant data is removed and if redundant data is removed.

To better evaluate the two systems and compare between their performances we constructed test cases to measure the detection rate, false positive and false negatives of both systems. Tables 7.10., 7.11. and 7.12. are examples of test cases that measure such criterions. We performed an incremental approach to test the behavior of the systems. For example, in table 7.10. we are comparing between the original lookahead pairs method and the enhanced version. We assumed 8 test cases, where we either have one mismatch (indicated by 1) or not (indicated by 0). If the CPU and memory concentrations are equal to 1 then there is a CPU or memory attack. We compared between two cases, either the identified mismatch is considered an intrusion or it is a new normal behavior. The same concept has been adapted for tables 7.11. and 7.12. In table 7.11., the mismatch threshold is 2 system calls and in table 7.12. the mismatch threshold is 3 system calls.

In general, if the sequence intention is normal and the system identified it as normal then this is correct output. If the sequence intention is an attack and the system identified it as an attack then this is a correct output. If the sequence intention is an attack and the system identified it as normal then this is considered as a false negative. If the system identified the sequence as normal and it was an attack then this is considered as a false positive.

			any mismatch is an intrusion					mismatches are normal				
CPU concentration	**Memory concentration**	**One system call mismatch**	**sequence intention**	**Original lookahead pairs method IDS**	**explanation of original**	**Lookahead pairs method enhanced with danger theory IDS**	**explanation of enhanced**	**sequence intention**	**Original lookahead pairs method IDS**	**explanation of original**	**Lookahead pairs method enhanced with danger theory IDS**	**explanation of enhanced**
0	0	0	N	N	OK	N	OK	N	N	OK	N	OK
0	0	1	A	A	OK	A	OK	N	A	FP	A	FP
0	1	0	A	N	FN	A	OK	A	N	FN	A	OK
0	1	1	A	A	OK	A	OK	A	A	OK	A	OK
1	0	0	A	N	FN	A	OK	A	N	FN	A	OK
1	0	1	A	A	OK	A	OK	A	A	OK	A	OK
1	1	0	A	N	FN	A	OK	A	N	FN	A	OK
1	1	1	A	A	OK	A	OK	A	A	OK	A	OK
Detection rate				62.5%		100%			50.0%		87.5%	
FP				0.0%		0.0%			12.5%		12.5%	
FN				37.5%		0.0%			37.5%		0.0 %	

Table 7.10. Performance comparison with mismatch threshold = 1 system call. N: normal behavior, A: attack, OK: produced correct output, FP: False positive, and FN: False negative.

The results obtained indicated that the enhanced system always performed better than or similar to the original lookahead pairs method based IDS with regard to detection rate, false positive and false negative rates. Both systems have less than 1 second testing time. When comparing between the original lookahead pairs method based IDS and the enhanced with iDC IDS, both have the same storage costs. However, if we are using the B cell functionalities then the cost will be doubled.

				any mismatch is an intrusion					mismatches are normal				
CPU concentration	**Memory concentration**	**another system call mismatch**	**One system call mismatch**	**sequence intention**	**Original lookahead pairs method IDS**	**explanation of original**	**Lookahead pairs method enhanced with danger theory IDS**	**explanation of enhanced**	**sequence intention**	**Original lookahead pairs method IDS**	**explanation of original**	**Lookahead pairs method enhanced with danger theory IDS**	**explanation of enhanced**
0	0	0	0	N	N	OK	N	OK	N	N	OK	N	OK
0	0	0	1	A	N	FN	N	FN	N	N	OK	N	OK
0	1	0	0	A	N	FN	A	OK	A	N	FN	A	OK
0	1	0	1	A	N	FN	A	OK	A	N	FN	A	OK
1	0	0	0	A	N	FN	A	OK	A	N	FN	A	OK
1	0	0	1	A	N	FN	A	OK	A	N	FN	A	OK
1	1	0	0	A	N	FN	A	OK	A	N	FN	A	OK
1	1	0	1	A	N	FN	A	OK	A	N	FN	A	OK
0	0	1	0	A	N	FN	N	OK	N	N	OK	N	OK
0	0	1	1	A	A	OK	A	FN	N	A	FP	A	FP
0	1	1	0	A	N	FN	A	OK	A	N	FN	A	OK
0	1	1	1	A	A	OK	A	OK	A	A	OK	A	OK
1	0	1	0	A	N	FN	A	OK	A	N	FN	A	OK
1	0	1	1	A	A	OK	A	OK	A	A	OK	A	OK
1	1	1	0	A	N	FN	A	OK	A	N	FN	A	OK
1	1	1	1	A	A	OK	A	OK	A	A	OK	A	OK
Detection rate					31.25		87.5			37.5		93.75	
FP					0		0			6.25		6.25	
FN					68.75		12.5			56.25		0	

Table 7.11. Performance comparison with mismatch threshold = 2 system calls. N: normal behavior, A: attack, OK: produced correct output, FP: False positive, and FN: False negative.

					any mismatch is an intrusion					mismatches are normal				
CPU concentration	**Memory concentration**	**another system call mismatch**	**another system call mismatch**	**One system call mismatch**	**sequence intention**	**Original lookahead pairs method IDS**	**explanation of original**	**Lookahead pairs method enhanced with danger theory IDS**	**explanation of enhanced**	**sequence intention**	**Original lookahead pairs method IDS**	**explanation of original**	**Lookahead pairs method enhanced with danger theory IDS**	**explanation of enhanced**
0	0	0	0	0	N	N	OK	N	OK	N	N	OK	N	OK
0	0	0	0	1	A	N	FN	N	FN	N	N	OK	N	OK
0	1	0	0	0	A	N	FN	A	OK	A	N	FN	A	OK
0	1	0	0	1	A	N	FN	A	OK	A	N	FN	A	OK
1	0	0	0	0	A	N	FN	A	OK	A	N	FN	A	OK
1	0	0	0	1	A	N	FN	A	OK	A	N	FN	A	OK
1	1	0	0	0	A	N	FN	A	OK	A	N	FN	A	OK
1	1	0	0	1	A	N	FN	A	OK	A	N	FN	A	OK
0	0	0	1	0	A	N	FN	N	FN	A	N	FN	N	FN
0	0	0	1	1	A	N	FN	N	FN	N	N	OK	N	OK
0	1	0	1	0	A	N	FN	A	OK	A	N	FN	A	OK
0	1	0	1	1	A	N	FN	A	OK	A	N	FN	A	OK
1	0	0	1	0	A	N	FN	A	OK	A	N	FN	A	OK
1	0	0	1	1	A	N	FN	A	OK	A	N	FN	A	OK
1	1	0	1	0	A	N	FN	A	OK	A	N	FN	A	OK
1	1	0	1	1	A	N	FN	A	OK	A	N	FN	A	OK
0	0	1	0	0	A	N	FN	A	OK	N	N	OK	N	OK
0	0	1	0	1	A	N	FN	N	FN	N	N	OK	N	OK
0	1	1	0	0	A	N	FN	A	OK	A	N	FN	A	OK
0	1	1	0	1	A	N	FN	A	OK	A	N	FN	A	OK
1	0	1	0	0	A	N	FN	A	OK	A	N	FN	A	OK
1	0	1	0	1	A	N	FN	A	OK	A	N	FN	A	OK
1	1	1	0	0	A	N	FN	A	OK	A	N	FN	A	OK
1	1	1	0	1	A	N	FN	A	OK	A	N	FN	A	OK
0	0	1	1	0	A	N	FN	N	FN	N	N	OK	N	OK
0	0	1	1	1	A	A	OK	A	OK	N	A	FP	A	FP
0	1	1	1	0	A	N	FN	A	OK	A	N	FN	A	OK
0	1	1	1	1	A	AA	OK	A	OK	A	A	OK	A	OK
1	0	1	1	0	A	N	FN	A	OK	A	N	FN	A	OK
1	0	1	1	1	A	A	OK	A	OK	A	A	OK	A	OK
1	1	1	1	0	A	N	FN	A	OK	A	N	FN	A	OK
1	1	1	1	1	A	A	OK	A	OK	A	A	OK	A	OK
Detection rate						15.625		84.375			28.175		93.75	
FP						0		0			3.125		3.125	
FN						84.375		15.625			68.75		3.125	

Table 7.12. Performance comparison with mismatch threshold = 3 system calls. N: normal behavior, A: attack, OK: produced correct output, FP: False positive, and FN: False negative.

7.8. Summary

Danger theory enhances the performance of system call based IDSs that are governed by a mismatch threshold. An attack may not produce mismatches or produce mismatches that do not exceed the mismatch threshold. However if an attack, including mismatches, is associated with unacceptable performance degradation (i.e. high CPU and memory usages) then the attack will be identified. In case the system identified mismatches but all monitored system conditions are normal then the system will not benefit from danger theory but will rely on it threshold to identify mismatches.

In our implementation only one object is generated for each class (cell) created in the system. For example, we have one B, one Th1, one Th2, one killer T, and one iDC and DC cells. This is different from other attempts to implement innate or adaptive based immunity based systems. The systems are usually populated by many instances or agents of such cells that circulate the system. Our system is not an agent based system. Rather, each type of cell in the immune system is represented with one object that carries out all the functionalities and duties of a population of this cell. This can be accomplished since all of them perform the same functionalities and only differ with respect to the monitored string or activity. As a result a database is associated with B and iDC classes holding the normal or abnormal strings (activities) to be monitored. Our system can be modified in many ways to handle more advanced and sophisticated tasks. For example, we can have different B cells each responsible for a specific application. We can also have more than one B cell responsible for portion of the whole problem of a specific application. This is beneficial if we have more than one processor and each B cell will run concurrently monitoring the same application.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this dissertation the concepts of adaptive immunity and in specific danger theory has been tested. The hypothesis of this dissertation was that incorporating properties of danger theory, the performance of lookahead pairs method which is an intrusion detection system that uses trails of system calls was enhanced. As explained earlier in tables 7.10. to 7.12., the lookahead pairs method based IDS enhanced with danger theory had better detection rate, better or similar false positive and false negative rates than the original lookahead pairs method. Both systems finished processing the input data file in less than one second. The storage requirement of the enhanced system with iDC is similar to the original lookahead pairs method IDS but the storage requirement of the system enhanced with B cell functionality is doubled.

Lookahead-pairs method has been previously proven to perform better with regard to storage requirements when compared with other intrusion detection systems such as sequence method and overlap-relationship method. However, this performance was tested on positive detectors. Positive detectors are generated by examining the normal behavior of the system and generating a database that hold this normal behavior. In general, with a specified and small number of patterns representing normal behavior, the negative detector set which represents the complement of normal behavior tend to be huge. As the number of negative detectors increase, the storage requirements of storing them also increase especially when using sequence and variable length detector methods that use data structures such as trees and linked lists to store such patterns. In this dissertation we identified that the characteristics of lookahead pairs method of storing pair relationships in a 2 dimensional array format does not required an additional storage.

Modeling danger theory functionalities can be achieved by instantiating one instance of each cell type. Rather than associating a different and specific signature string to each B or iDC cell of which it uses to identify intrusions, our system instantiated one object of each cell type. The B and iDC cells are associated with a database of all possible normal behavior signatures.

The original sequence method, lookahead pairs method and variable length with overlap relationship method were unable to detect system-call-denial-of-service-attack. Enhancing them with danger theory will enable them to identify such attack since consuming system resources will indicate an intrusion and will be identified by the enhanced system.

Danger theory is currently investigated to solve many security and non-security related problems. Exploring immune system concepts and theory are exciting and interesting. In this dissertation we deployed danger theory to enhance an IDS. Our future work will continue in the following fields:

- Enhancing the performance of our system and detect intrusions that deviate monitored parameters.
- Implement danger theory inspired intrusion detection system for hand held devices.

- Enhance other intrusion detection systems with danger theory concepts and in particular sequence and variable length with overlap relationship methods.
- Investigate other suitable parameters to be used to indicate dangerous signals to the danger theory cells.
- Investigate techniques to reduce the cost of storing pattern databases by using different data structures.
- Implement an on – line intrusion detection system performing the functionalities tested in the off-line systems implemented in this dissertation.

This dissertation explored the different mechanisms employed to detect host based intrusions by examining sequences of system calls produced by a specific application. Artificial immune systems and in specific danger theory concepts were employed to enhance the performance of lookahead pairs method and it was successful in outperforming the original form of the IDS. Better detection rate and lower or similar false positive and false negative rates were achieved.

REFERENCES

[Aickelin 2004] U. Aickelin. Artificial Immune Systems: A new paradigm For Heuristic Decision Making. Invited Keynote Talk, Annual Operational Research Conference 46, York, UK. 2004.

[Aickelin and Cayzer 2002] U. Aickelin and S. Cayzer S. The Danger Theory and Its Application to AIS, 1st International Conference on AIS, pp 141-148. 2002.

[Aickelin and Dasgupta 2005] U. Aickelin and D. Dasgupta. Artificial Immune Systems Tutorial. To appear in Introductory Tutorials in Optimization, Decision Support and Search Methodology (eds. E. Burke and G. Kendall), Kluwer. 2005.

[Aickelin et al. 2003] U. Aickelin, P. Bentley, S. Cayzer, J. Kim and J. McLeod. Danger Theory: The Link between AIS and IDS. Proceedings ICARIS-2003, 2nd International Conference on Artificial Immune. 2003.

[Aickelin, Greensmith and Twycross 2004] U. Aickelin, J. Greensmith and J. Twycross. Immune system approaches to intrusion detection-a review. In: Proc. ICARIS-04, 3rd Int. Conf. on Artificial Immune Systems (Catania, Italy), Lecture Notes in Computer Science, Vol. 3239, pp. 316--329, Springer, Berlin. 2004.

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

[Axelsson 1999] Stefan Axelsson. Research in intrusion-detection systems: a survey. Technical report 98-17. Department of Computer Engineering, Chalmers University of Technology, December 1998.

[Axelsson 2000] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report No 99-15, Chalmers University of Technology, Sweden, 2000.

[Ayara et al. 2002] M. Ayara, J. Timmis, R. D. Lemos, D. Castro, and R. Duncan. Negative Selection: How to Generate Detectors. In: J. Timmis and P. Bentley (eds.): Proceedings of 1st International Conference on Artificial Immune Systems (ICARIS). Canterbury, UK, pp. 89-98. 2002.

[Balthrop et al. 2002] Justin Balthrop, Fernando Esponda, Stephanie Forrest and Matthew Glickman. Coverage and Generalization in an Artificial Immune System. Proceedings of the 2002 Genetic and Evolutionary Computation Conference (GECCO 2002).

[Balthrop, Forrest and Glickman 2002] J. Balthrop, S. Forrest and M. Glickman. Revisiting lisys: Parameters and normal behavior. Proc. of the Congress on Evolutionary Computation, pages 1045–1050. In CEC-2002.

[Begnum and Burgess 2003] K. Begnum and M. Burgess. A scaled, immunological approach to anomaly countermeasures (combining ph with cfengine). Integrated Network Management, pages 31-42, 2003.

[Bentley, Greensmith and Ujjin 2005] P. Bentley, J. Greensmith, and S. Ujjin. Two ways to grow tissue for artificial immune systems. In International Conference on Artificial Immune Systems (ICARIS), LNCS 3627, pages 139–152, 2005.

[Burgess 1998] M Burgess. Computer immunology. In Proc. of the Systems Administration Conference (LISA-98), pages 283--297, 1998.

[Burgess 2000] M. Burgess. Evaluating cfegine's immunity model of site maintenance. In Proceeding of the 2nd SANE System Administration Conference (USENIX/NLUUG), 2000.

[Burgess 2001] M. Burgess. Recent developments in cfengine. In Proceedings of the 2nd Unix.nl conference, Netherlands, 2001.

[Burgess 2002] M. Burgess. Two dimensional time-series for anomaly detection and regulation in adaptive systems. In M. Feridum et al., editor, Proceedings of 13th IFIP/IEEE International Workshop on Distributed System, Operations and Management (DSOM 2002), volume 2506 of Lecture Notes in Computer Science, pages 169-180. Springer-Verlag, 2002.

[Burgess 2004a] M. Burgess. Configurable immunity for evolving human-computer systems. Science of Computer Programming, 51:197-213, 2004.

[Burgess 2004b] M. Burgess. Principle components and importance ranking of distributed anomalies. Machine Learning, 58:217-230, 2004.

[Cayzer and Aickelin 2002a] S. Cayzer and U. Aickelin. A Recommender System based on the Immune Network. Proceedings CEC, pp 807-813. 2002.

[Cayzer and Aickelin 2002b] S. Cayzer and U. Aickelin. On the Effects of Idiotypic Interactions for Recommendation Communities in Artificial Immune Systems. Research Report BICAS-2002-15, HP Labs, Bristol, UK. 2002

[Cayzer and Aickelin 2005] Steve Cayzer1 and Uwe Aickelin. A Recommender System based on Idiotypic Artificial Immune Networks. Submitted & Under Review by JMMA.

[Chao and Forrest 2002] D. L. Chao and S. Forrest. Information Immune Systems. In Proceedings of the First International Conference on Artificial Immune Systems (ICARIS), pp. 132-140 2002.

[D'haeseleer 1996] P. D'haeseleer. An immunological approach to change detection: theoretical results. In Proceedings of the 9th IEEE computer Security Foundations Workshop. IEEEE computer Society Press, 1996.

[D'haeseleer, Forrest and Helman, 1996] P. D'haeseleer, S. Forrest, and P. Helman. An immunological approach to change detection: algorithms, analysis and implications. In proceedings of the 1996 IEEE Symposium on Computer Security and Privacy. IEEE Press, 1996.

[Dain and Cunnigham 2001] O. Dain and R. K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In ACM Workshop on Data Mining for Security Applications, pages 1-13, 2001.

[Dasgupta 1999] D. Dasgupta. Immunity-based intrusion detection systems: A general framework. In Proc. of the 22nd National Information Systems Security Conference (NISSC), October 1999.

[Dasgupta 2004] D. Dasgupta. Immuno-Inspired Autonomic System for Cyber Defense. Computer Science Technical Report, May, 2004.

[Dasgupta and Attoh-Okine 1997] D. Dasgupta and N. Attoh-Okine. Immunity-based systems: A survey. In Proceedings of the IEEE International Conference on

Systems, Man, and Cybernetics, pp. 363-374, Orlando, Florida, October 12-15 1997.

[Dasgupta and Gonzalez 2002] D. Dasgupta and F. Gonzalez. An Immunity-Based Technique to Characterize Intrusions in Computer Networks. IEEE Trans. Evolution Computer. Vol. 6; 3, pp 1081-1088. 2002.

[De Paula, De Castro and De Geus 2004] F. S. de Paula, L. N. de Castro, and P. L. de Geus. An intrusion detection system using ideas from the immune system. In Proceeding of IEEE Congress on Evolutionary Computation (CEC-2004), pages 1059-1066, Portland, Oregon, USA, June 2004.

[Debar et al. 1998] Hervé Debar , Marc Dacier , Mehdi Nassehi , Andreas Wespi, Fixed vs. Variable-Length Patterns for Detecting Suspicious Process Behavior. Proceedings of the 5th European Symposium on Research in Computer Security, p.1-15, September 16-18, 1998

[Debar, Cacier and Wespi 2000] H. Debar, M. Dacier, and A. Wespi. A revised taxonomy of intrusion-detection systems. Annales des Telecommunications, 55:83-100, 2000.

[Denning 1987] D. E. Denning. An Intrusion Detection Model. IEEE Transactions on Software Engineering, 13(2):222–232, 1987.

[Dozier et al. 2004] G. Dozier, D. Brown, J. Hurley, and K. Cain. Vulnerability analysis of immunity-based intrusion detection systems using evolutionary hackers. In K. Deb et al, editor, Genetic and Evolutionary Computation - GECCO-2004, Part I, volume 3102 of Lecture Notes in Computer Science, pages 263-274, Seattle, WA, USA, 26-30 June 2004. ISGEC, Springer-Verlag.

[Eiben, Hinterding and Michalewicz 1999] A. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. IEEE Transactions on Evolutionary Computation, 3:124-141, 1999.

[Endler 1998] D. Endler. Intrusion Detection Applying Machine Learning to Solaris Audit Data. In Proc. of the IEEE Annual Computer Security Applications Conference, pages 268–279, Scottsdale, AZ, 1998.

[Esponada, Forrest and Helman 2003] F. Esponda, S. Forrest, and P. Helman. The crossover closure and partial match detection. In J Timmis, P Bentley, and E Hart, editors, Proceedings of the 2nd International Conference on Artificial Immune Systems (ICARIS'-03), volume 2787 of Lecture Notes in Computer Science, pages 249-260, Edinburgh, UK, September 2003. Springer-Verlag.

[Esponda and Forrest 2002] F. Esponda and S. Forrest. Defining Self: Positive and Negative detection . Technical Report TR-CS-2002-03, University of New Mexico, 2002.

[Esponda, Forrest and Helman 2004] Fernando Esponda, Stephanie Forrest and Paul Helman. A formal Framework for Positive and Negative Detection Schemes. IEEE Transactions on Systems. Man and Cybernetics 2004.

[Forrest and Hofmeyr 2001a] Stephanie Forrest and Steven A. Hofmeyr. Immunology as information processing, In Design Principles for the Immune System and Other Distributed Autonomous Systems, edited by L.A. Segel and I. Cohen. Santa Fe Institute Studies in the Sciences of Complexity. New York: Oxford University Press. (2001).

[Forrest and Hofmeyr 2001b] S. Forrest, S. Hofmeyr. Engineering an Immune System. Graft, Vol. 4, No. 5, 2001, 5-9

[Forrest et al. 1994] Stephanie Forrest, Alan S Perelson, Lawrence Allen and Rajesh Cherukuri. Self-Non-self Discrimination in a Computer. Proceedings of the IEEE Symposium on Research in Security and Privacy, IEEE Press(1994).

[Forrest et al. 1996] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In Proceedings of 1996 IEEE Symposium on Computer Security and Privacy, pp. 120-128 (1996).

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

[Fraser, Badger and Feldman 1999] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In Proc. of the IEEE Symposium on Security and Privacy, pages 2–16, 1999.

[Gao, Reiter and Song 2004a] D. Gao, M. K. Reiter, and D. Song. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In Proc. of the ACM Conference on Computer and Communications Security, pages 318–329, 2004.

[Gao, Reiter and Song 2004b] D. Gao, M. K. Reiter, and D. Song. On Gray-Box Program Tracking for Anomaly Detection. In Proc. of the 13th USENIX Security Symposium, pages 103–118, San Diego, CA, August 2004.

[Garfinkel 2003] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools. In Proc. of the Network and Distributed Systems Security Symposium, pages 162–177, 2003.

[Garfinkel, Pfaff and Rosenblum 2004] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In Proc. of the Network and Distributed Systems Security Symposium, pages 52–69, 2004.

[Ghosh, Schwartzbard and Schatz 1999] A. K. Ghosh, A. Schwartzbard and M. Schatz. Learning Program Behavior Profiles for Intrusion Detection. 1st USENIX Workshop on Intrusion Detection & Networking Monitoring, 1999.

[Goldberg 1996] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for un-trusted helper applications: Confining the wily hacker. In Proc. of the 6th USENIX Security Symposium, pages 1–13, San Jose, CA, July 1996.

[Gomez, Gonzalez and Dasgupta 2003] J. Gomez, F. Gonzalez, and D. Dasgupta. An immuno-fuzzy approach to anomaly detection. In proceedings of the 12th IEEE International Conference on Fuzzy Systems (FUZZIEEE), volume 2, pages 1219-1224, May 2003.

[Gonzalez 2003] F. Gonzalez. A Study of Artificial Immune Systems Applied to Anomaly Detection. PhD thesis, The University of Memphis, May 2003.

[Gonzalez and Cannady 2004] L. J. Gonzalez and J. Cannady. A self-adaptive negative selection approach for anomaly detection. In Proceedings of the 2004 Congress of Evolutionary Computation (CEC-2004), pages 1561-1568. IEEE Computer Society, 2004.

[Gonzalez and Dasgupta 2002] F. Gonzalez and D. Dasgupta. An immunogenetic technique to detect anomalies in network traffic. In Proceedings of the genetic and evolutionary computation conference, GECCO 2002.

[Gonzalez and Dasgupta 2003] F. Gonzalez, and D. Dasgupta. Anomaly detection using real-valued negative selection. In Genetic Programming and Evolvable Machines, Vol. 4, pp. 383--403. 2003.

[Gonzalez et al. 2005] F. A. Gonzalez, J. C. Galeano, D. A. Rojas, and A. Veloza-Suan. Discriminating and visualizing anomalies using negative selection and self-organizing maps. In H.-G. Beyer et al., editor, GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, volume 1, pages 297-304, Washington DC, USA, 25-29, June 2005. ACM SIGEVO (formerly ISGEC), ACM Press.

[Gonzalez, Dasgupta and Kozma 2002] F. Gonzalez, D. Dasgupta, , and R. Kozma. Combining negative selection and classification techniques for anomaly detection. In IEEE, editor, Proceedings of the Congress on Evolutionary Computation (CEC-2002), pages 705-710, Honolulu, HI, May 2002.

[Gonzalez, Dasgupta and Nino 2003] F. Gonzalez, D. Dasgupta, and L. F. Nino. A randomized real-valued negative selection algorithm. In J Timmis, P Bentley, and E Hart, editors, Proceedings of the 2nd International Conference on Artificial Immune Systems (ICARIS-2003), volume 2787 of Lecture Notes in Computer Science, pages 261-272, Edinburgh, UK, September 2003. Springer.

[Greensmith and Aickelin 2006] J. Greensmith and U. Aickelin. Dendritic Cells for Real-Time Anomaly Detection. Proceedings of the Workshop on Artificial Immune Systems and Immune System Modeling (AISB 2006), pp 7-8, Bristol, UK. 2006.

[Greensmith, Aickelin and Cayzer 2005] J. Greensmith, U. Aickelin and S. Cayzer. Introducing Dendritic Cells as a Novel Immune-Inspired Algorithm for Anomaly Detection. Research Report HPL-2005-117, HP Labs, Bristol, UK. 2005.

[Greensmith, Aickelin and Twycross 2006] J. Greensmith, U. Aickelin and J. Twycross, J. Articulation and Clarification of the Dentric Cell Algorithm. Proceedings of the 5th International Conference on Artificial Immune Systems. (ICARIS 2006) LNCS 4163, pp 404-417. Oeiras, Portugal. 2006.

[Greensmith, Twycross and Aickelin 2006] J. Greensmith, J. Twycross and U. Aickelin. Dendritic Cells for Anomaly Detection. Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2006), pp XXX, Vancouver, Canada. 2006.

[GSWT] GSWT homepage. http://opensource.nailabs.com/wrappers/, 2007.

[Hang and Dai 2004] X. Hang and H. Dai. Constructing detectors in schema complementary space for anomaly detection. In K. Deb et al., editor, Proceedings of GECCO'2004, volume 3102 of Lecture Notes in Computer Science, pages 275-286. Springer-Verlag, 2004.

[Hang and Dai 2005] X. Hang and H. Dai. Applying both positive and negative selection to supervised learning for anomaly detection. In H.-G. Beyer et al., editor, GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, volume 1, pages 345-352, Washington DC, USA, 25-29 June 2005. ACM SIGEVO (formerly ISGEC), ACM Press.

[Harmer et al. 2002] P. K. Harmer, P. D. Williams, G. H. Gunsch, and G. B. Lamont. An artificial immune system architecture for computer security applications. IEEE Transactions on Evolutionary Computation, 6(3):252-280, June 2002.

[Hofmeyr 1999] Steven Hofmeyr. An immunological model of distributed detection and its application to computer security. PhD thesis, University Of New Mexico, 1999.

[Hofmeyr 2000] S. A. Hofmeyr. An Interpretative Introduction to the Immune System. In: Segel, L.A., Cohen, I.R. (Eds.), Design Principles for the Immune System and Other Distributed Autonomous Systems, Oxford University Press, New York. pp. 3-26. 2000.

[Hofmeyr and Forrest 1998] S. Hofmeyr and S. Forrest. Intrusion Detection using Sequences of System Calls. Journal of Computer Security, 6(3):151–180, 1998.

[Hofmeyr and Forrest 1999a] S. Hofmeyr and S. Forrest. Immunity by design: An artificial immune system. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), pages 1289-1296, San Francisco, CA, 1999. Morgan-Kaufmann.

[Hofmeyr and Forrest 1999b] Steven A. Hofmeyr and Stephanie Forrest. Immunizing Computer Networks: Getting All the Machines in Your Network to Fight the Hacker Disease. Proc. 1999 IEEE Symposium on Security and Privacy.

[Hofmeyr and Forrest 2000] Hofmeyr S. and Forrest S. Architecture for an Artificial Immune System. Evolutionary Computation, 8,(4), 443-473. 2000.

[Hofmeyr, Forrest and Somayaji 1998] S. A. Hofmeyr, S. Forrest and A. Somayaji. Intrusion detection using sequences of system calls. Journal of Computer Security, 6 (1998), 151--180.

[Hou and Dozier 2005] Haiyu Hou and Gerry Dozier. Immunity-based intrusion detection system design, vulnerability analysis, and GENERTIA's genetic arms race.

Symposium on Applied Computing archive Proceedings of the 2005 ACM symposium on Applied computing . 2005.

[Janeway et al. 2005] C. A. Janeway, P. Travers, M. Walport, and M. Shlomchik. Immuno-biology: The Immune System in Health and Disease. Garland Publishing.Available online at http://www.ncbi.nlm.nih.gov/books/, 6th edition, 2005.

[janus] janus homepage. http://www.cs.berkeley.edu/~daw/janus/,2007.

[Ji and Dasgupta 2004] Z. Ji and D. Dasgupta. Augmented negative selection algorithm with variable-coverage detectors. In Proceedings of Congress on Evolutionary Computation (CEC-04), pages 1081-1088, Portland, Oregon (U.S.A.), June 2004.

[Ji and Dasgupta 2005] Z. Ji and D. Dasgupta. Estimating the detector coverage in a negative selection algorithm. In H.-G. Beyer et al., editor, GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, volume 1, pages 281-288, Washington DC, USA, 25-29 June 2005. ACM SIGEVO (formerly ISGEC), ACM Press.

[Jiang, Hua and Oh 2003] N. Jiang, K. Hua and J-H. Oh. Exploiting Pattern Relationship for Intrusion Detection. Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03). IEEE. 2003.

[Jiang, Hua and Sheu] N. Jiang, K. Hua and S. Sheu. Considering Both Intra- Pattern and Inter-Pattern Anomalies for Intrusion Detection. IEEE. 2002.

[Kephart 1994] J. Kephart. A biologically inspired immune system for computers. In Proceedings of the Fourth International Workshop on Synthesis and Simulation of Living Systems, Artificial Life IV, pages 130-139, 1994.

[Kephart et al. 1998] J. O. Kephart, G. B. Sorkin, M. Swimmer, and S. R. White. Blueprint for a Computer Immune System. pages 241-261. Artificial Immune Systems and Their Applications. Springer-Verlag, 1998.

[Kim 2002] J. Kim. Integrating Artificial Immune Algorithms for Intrusion Detection. PhD Thesis, University College London. 2002.

[Kim and Bentley 1999a] J. Kim, and P. Bentley. An Artificial Immune Model for Network Intrusion Detection. 7th European Congress on Intelligent Techniques and Soft Computing (EUFIT'99). 1999.

[Kim and Bentley 1999b] J. Kim and P. Bentley. Negative selection and niching by an artificial immune system for network intrusion detection. In GECCO-99 Proceedings, p. 149--158, 1999.

[Kim and Bentley 1999c] J. Kim and P. Bentley. The artificial immune model for network intrusion detection. In Proc. of European Congress on Intelligent Techniques and Soft Computing (EUFIT '99), Aachen, Germany, September 1999.

[Kim and Bentley 1999d] J. Kim and P. Bentley. The Human Immune System and Network Intrusion Detection. In Proceedings of the 7th European Congress on Intelligent Techniques and Soft Computing (EUFIT'99). 1999.

[Kim and Bentley 2001a] Jungwon Kim and Peter J. Bentley. An evaluation of negative selection in an artificial immune system for network intrusion detection. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001.

[Kim and Bentley 2001b] J. Kim and P. J Bentley. Towards an artificial immune system for network intrusion detection: An investigation of clonal selection with a negative selection operator. In Proceeding of the Congress on Evolutionary Computation (CEC-2001), Seoul, Korea, pages 1244-1252, 2001.

[Kim and Bentley 2002a] J. Kim and P. Bentley. A Model of Gene Library Evolution in the Dynamic Clonal Selection Algorithm. Proceedings of the First International Conference on Artificial Immune Systems (ICARIS) Canterbury, pp.175-182, September 9-11, 2002.

[Kim and Bentley 2002b] J. Kim and P. Bentley. Immune Memory in the Dynamic Clonal Selection Algorithm. Proceedings of the First International Conference on Artificial Immune Systems (ICARIS) Canterbury, pp.57-65, September 9-11, 2002.

[Kim and Bentley 2002c] J. Kim, P. Bentley. Towards an AIS for Network Intrusion Detection: An Investigation of Dynamic Clonal Selection. The Congress on Evolutionary Computation 2002, pp 1015-1020.

[Kim et al. 2005a] J. Kim, J. Greensmith, J. Twycross and U. Aickelin. Malicious Code Execution Detection and Response Immune System inspired by the Danger Theory. Adaptive and Resilient Computing Security Workshop (ARCS 2005), Santa Fe, USA

[Kim et al. 2005b] J. Kim, W. Wilson, U. Aickelin, and J. McLeod. Cooperative automated worm response and detection immune algorithm (cardinal) inspired by t-cell immunity and tolerance. In C. Jacob, M. J. Pilat, P. J.Bentley, and J. Timmis, editors, Proceeding of the 4th International Conference on Artificial Immune

Systems (ICARIS-2005), volume 3627 of Lecture Notes in Computer Science, pages 168-181, Banff, Alberta, Canada, August 2005. Springer.

[Kim et al. 2007] J. Kim, P. Bentley, U. Aickelin, J. Greensmith, G. Tedesco and J. Twycross. Immune System Approaches to Intrusion Detection - A Review. Natural Computing, Springer, in print, pp XXX. 2007.

[Ko et al. 2000] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In Proc. of the 9th USENIX Security Symposium, pages 145–146, Denver, Colorado, August 2000.

[Ko, Fink and Levitt 1994] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In Proc. of the IEEE Annual Computer Security Applications Conference, pages 134–144, Orlando, FL, 1994.

[Ko, Ruschitzka and Levitt 1997] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification based approach. In Proc. of the IEEE Symposium on Security and Privacy, pages 175–187, 1997.

[Kruegel et al. 2003] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In Proc. of the 8th European Symposium on Research in Computer Security, pages 326–343, Gjovik, Norway, October 2003.

[Lane and Brodley 1997] T. Lane and C. E. Brodley. Sequence Matching and Learning in Anomaly Detection for Computer Security. AAAI Workshop: AI Approaches to Fraud Detection and Risk Management, pp.49, 1997.

[Lane and Brodley 1999] T. Lane, and C. E. Brodley. Temporal Sequence Learning and Data Reduction for Anomaly Detection. ACM Trans. Information & System Security, vol. 2, no. 3, pp. 295-331, 1999.

[Le Boudec and Sarafijanovic 2003] J. Le Boudec and S. Sarafijanovic. An artificial immune system approach to misbehavior detection in mobile ad-hoc networks. Technical Report IC/2003/59, Ecole Polytechnique Federale de Lausanne, 2003.

[Le Boudec and Sarafijanovic 2004] J. Le Boudec and S. Sarafijanovic. An artificial immune system approach to misbehavior detection in mobile ad-hoc networks. In Proceedings of Bio-ADIT 2004 (The First International Workshop on Biologically Inspired Approaches to Advanced Information Technology), pages 96-111, Lausanne, Switzerland, January 2004.

[Leach and Tedesco 2003] J. Leach and G. Tedesco. Firestorm network intrusion detection system. Firestorm Documentation, 2003.

[Lee and Stolfo 1998] W. Lee and S. J. Stolfo. Data Mining Approaches for Intrusion Detection. 7th USENIX Security Symposium, 1998.

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

[Lee, Stolfo and Mok 1999] W. Lee, S. J. Stolfo and K. Mok. A Data Mining Framework for Building Intrusion Detection Models. IEEE Symposium on Security & Privacy, 1999.

[Lodish et al. 1999] H. Lodish, A. Berk, P. Matsudaira, C. A. Kaiser, M. Krieger, M. P. Scott, L. Zipursky, and J. Darnell. Molecular Cell Biology. W. H. Freeman and Co.. Available online at http://www.ncbi.nlm.nih. gov/books/, 4th edition, 1999.

[LTT] Linux Trace Toolkit (LTT) homepage. http://www.opersys.com/ LTT/, 2007.

[Maniatty et al. 2005] W. A. Maniatty, A. Baykal, V. Aggarwal, J. Brooks, A. Krymer, and S. Maura. A Linux kernel auditing tool for host-based intrusion detection. In Proc. of the IEEE Annual Computer Security Applications Conference, pages 307–313, Tucson, AZ, 2005.

[Matzinger 1994] P. Matzinger. Tolerance, Danger and the Extended Family. Annual Review of Immunology, 12:991-1045, 1994.

[Matzinger 2002] P. Matzinger. The Danger Model: A Renewed Sense of Self. Science 296: 301-305. 2002.

[Medzhitov and Janeway 2002] R. Medzhitov and C. A. Janeway. Decoding the patterns of self and non-self by the innate immune system. Science, 296(5566):298-300. 2002.

[Mell et al. 2003] P. M. Mell, V. C. Hu, R. Lippmann, J. Haines, and M. Zissman. An Overview of Issues in Testing Intrusion Detection Systems. Technical Report NIST Interagency Reports 7007, National Institute of Standards and Technology, July 2003.

[Michael and Ghosh 2000] C. Michael, A. Ghosh. Using Finite Automata to Mine Execution Data for Intrusion Detection: A Preliminary Report. RAID 2000, LNCS 1907, pp.66-79, 2000.

[Morrison and Aickelin 2002] T. Morriso and U. Aickelin. An AIS as a Recommender System for Web Sites. 1st International Conference on AIS, pp 161-169. 2002.

[Neal and Timmis 2005] M. Neal and J. Timmis. Once More Unto the Breach: Towards Artificial Homeostasis. In L. N. D. Castro and F. J. V. Zuben, editors, Recent Developments in Biologically Inspired Computing, pages 340– 365. Idea Publishing Group, 2005.

[Ning et al. 2004] P. Ning, D. Xu, C. G. Healey, and R. S. Amant. Building attack scenarios through integration of complementary alert correlation method. In NDSS, 2004.

[Ourston et al. 2002] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of Hidden Markov Models to Detecting Multistage Network Attacks. Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03). IEEE. 2002.

[Pagnoni and Visconti 2005] A. Pagnoni and A. Visconti. An innate immune system for the protection of computer networks. In Proc. of the 4th International Symposium on Information and Communication Technologies, pages 63–68. Trinity College Dublin, 2005.

[Provos 2003] N. Provos. Improving Host Security with System Call Policies. In Proc. of the 12th USENIX Security Symposium, pages 257–272, Washington, D.C., August 2003.

[Rigoutsos and Floratos 1998] I. Rigoutsos, and Aristidis Floratos. Combinatorial Pattern Discovery in Biological Sequences: The TEIRESIAS Algorithm. Bioinformatics, 14(1): 55-67, 1998.

[Sarafaijanovic and Le Boudec 2004] S. Sarafijanovic and J. Le Boudec. An artificial immune system for misbehavior detection in mobile ad-hoc networks with virtual thymus, clustering, danger signal and memory detectors. In Proceedings of the 3rd International Conference on Artificial Immune Systems (ICARIS'-04), pages 342-356, Catania, Italy, September 2004.

[Sarafijanovic and Le Boudec 2003] S. Sarafijanovic and J. Le Boudec. An artificial immune system approach with secondary response for misbehavior detection in mobile ad-hoc networks. Technical Report IC/2003/65, Ecole Polytechnique Federale de Lausanne, 2003.

[Sarafijanovic and Le Boudec 2005] S. Sarafijanovic and J.-Y. Le Boudec. An Artificial Immune System Approach with Secondary Response for Misbehavior Detection in Mobile Ad-Hoc Networks. IEEE Transactions on Neural Networks, Special Issue on Adaptive Learning Systems in Communication Networks, 16(5):1076–1087, 2005.

[Sekar et al. 2001] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In Proc. of the IEEE Symposium on Security and Privacy, pages 144–155, 2001.

[Sekar, Bowen and Segal 1999] R. Sekar, T. Bowen, and M. Segal. On Preventing Intrusions by Process Behavior Monitoring. In Proc. of the USENIX Workshop on Intrusion Detection and Network Monitoring, pages 29–40, Santa Clara, CA, April 1999.

[Shapiro, Lamont and Peterson 2005] J. M. Shapiro, G. B. Lamont, and G. L. Peterson. An evolutionary algorithm to generate hyper-ellipsoid detectors for negative selection. In H.-G. Beyer et al., editor, GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, volume 1, pages 337-344, Washington DC, USA, 25-29, June 2005. ACM SIGEVO (formerly ISGEC), ACM Press.

[Singh and Nair 2005] C. T. Singh and S. B. Nair. An Artificial Immune System for a Multi Agent Robotics System. In Proc. of the 4th World Enformatika International Conference on Automation Robotics and Autonomous Systems (ARAS 2005), pages 308–311, 2005.

[Smith, Forrest and Perelson 1993] R. E. Smith, S. Forrest, and A. S. Perelson. Searching for diverse, cooperative population with genetic algorithms. Evolutionary Computation, 1(2):127-149, 1993.

[snare] snare homepage. http://www.intersectalliance.com/projects/ Snare/, 2007.

[Somayaji 2002] A. Somayaji. Operating System Stability and Security through Process Homeostasis. PhD thesis, University Of New Mexico, 2002.

[Somayaji and Forrest 2000] A. Somayaji and S. Forrest. Automated response using system-call delays. In Proc. of the 9th USENIX Security Symposium, pages 185–198, Denver, CO, August 2000.

[Somayaji, Hofmeyr and Forrest 1998] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a Computer Immune System. 1997 New Security Paradigms Workshop, pp. 75-82, ACM (1998).

[Stepney et al. 2005] S. Stepney, R. Smith, J. Timmis, A. Tyrrell, M. Neal, and A. Hone. Conceptual Frameworks for Artificial Immune Systems. International Journal of Unconventional Computing, 1(3):315–338, 2005.

[Stibor 2006] T. Stibor. On the Appropriateness of Negative Selection for Anomaly Detection and Network Intrusion Detection. PhD thesis, Darmstadt University of Technology, Germany, 2006.

[Stibor et al. 2005] T. Stibor, P. Mohr, J. Timmis, and C. Eckert. Is negative selection appropriate for anomaly detection. In H.-G. Beyer et al., editor, GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, volume 1, pages 321-328, Washington DC, USA, 25-29, June 2005. ACM SIGEVO (formerly ISGEC), ACM Press.

[Stibor, Timmis and Eckert 2005] T. Stibor, J. Timmis, and C. Eckert. On the appropriateness of negative selection defined over hamming shape-space as a network intrusion detection system. In Proceedings of the Congress on Evolutionary Computation (CEC-2005), pages 995-1002, Edinburgh, UK, September 2005. IEEE Press.

[Stillerman, Marceau and Stillman 1999] M. Stillerman, C. Marceau, and M. Stillman. Intrusion detection for distributed application. Communications of the ACM, 42(7):62-69, July 1999.

[strace] strace homepage. http://sourceforge.net/projects/strace/, 2007.

[syscalls] syscalls. Linux man page (2), 2007.

[systrace] http://www.systrace.org [September 11, 2006]

[Tandon and Chan 2003] G. Tandon and P. Chan. Learning rules from system call arguments and sequences for anomaly detection. In ICDM Workshop on Data Mining for Computer Security (DMSEC), pages 20--29, 2003.

[Tandon and Chan 2005] G. Tandon and P. K. Chan. Learning Useful System Call Attributes for Anomaly Detection. In Proc. of the 18th International FLAIRS Conference, pages 405–411, 2005.

[Tandon, Chan and Mitra 2004] G. Tandon, P. K. Chan, and D. Mitra. MORPHEUS: motif oriented representations to purge hostile events from unlabeled sequences. In Proc. of the ACM workshop on Visualization and Data Mining for Computer Security, pages 16–25. ACM Press, 2004.

[The Danger Project] The Danger Project website. http://www.dangertheory.com/, 2007.

[Twycorss and Aickelin 2005] J. Twycross and U. Aickelin. Towards a Conceptual Framework for Innate Immunity. Proceedings of the 4th International Conference on Artificial Immune Systems (ICARIS 2005), LNCS 3627, pp 112-125, Banff, Canada. 2005.

[Twycorss and Aickelin 2006a] J. Twycross and U. Aickelin. Experimenting with innate immunity. Proceedings of the Workshop on Artificial Immune Systems and Immune System Modeling (AISB 2006), pp 18-19, Bristol, UK. 2006.

[Twycorss and Aickelin 2006b] J. Twycross and U. Aickelin. Libtissue - implementing innate immunity. Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2006), pp XXX, Vancouver, Canada. 2006.

[Twycross 2007]. Jamie Paul Twycross. Integrated Innate and Adaptive Artificial Immune Systems Applied to Process Anomaly Detection. PhD thesis. University of Nottingham, 2007.

[Valdes and Skinner 2001] A. Valdes and K. Skinner. Probabilistic alert correlation. In RAID '00: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection, pages 54-68. Springer-Verlag, 2001.

[Wagner 1999] D. Wagner. Janus: an Approach for Confinement of Un-trusted Applications. Technical Report CSD-99-1056, University of California at Berkeley, December 1999.

[Wagner and Dean 2001] D. Wagner and D. Dean. Intrusion detection via static analysis. In Proc. of IEEE Symposium on Security and Privacy, pages 156–169, 2001.

[Warrender, Forrest and Pearlmutter] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions using system calls: alternative data models. IEEE Symposium on Security and Privacy. May 1999.

[Wespi, Dacier and Debar 1999] A. Wespi, M. Dacier, and H. Debar. An intrusion detection system based on the teiresias pattern discovery algorithm. In EICAR Annual Conference Proceedings, pp. 1-15, 1999.

[Wespi, Dacier and Debar 2000]Andreas Wespi , Marc Dacier , Hervé Debar, Intrusion Detection Using Variable-Length Audit Trail Patterns. Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection, p.110-129, October 02-04, 2000

[Xie et al. 2004] Y. Xie, H. Kim, D. R. O'Hallaron, M. K. Reiter, and H. Zhang. Seurat: A pointillist approach to anomaly detection. In RAID, pages 238-257, 2004.

[Yaghmour and Dagenais 2000] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In Proc. of the 9th USENIX Annual Technical Conference, pages 13–26, San Diego, CA, June 2000.

[Yeung and Ding 2003] D. Y. Yeung and Y. Ding. Host-based intrusion detection using dynamic and static behavioral models. Pattern Recognition, 36(1):229– 243, 2003.

APPENDICES

APPENDIX A

SEQUENCE METHOD BASED IDS SAMPLE CODE

```
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//SEQUENCE METHOD BASED IDS
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

//-----------------------------------------------------------------------------------------------------------------
//TESTING
//-----------------------------------------------------------------------------------------------------------------
void testing( Seqhash<int> table[NUM_SYS_CALLS])
{
	ofstream outPrintFile2 ("sequence_results.txt", ios::out);
		if (!outPrintFile2)
		{
			cerr <<"int files names file could not be opened" <<endl;
			exit(1);
		}
	time_t begining_time;
	time(&begining_time);
	outPrintFile2 << "Begining Time= "<<begining_time<<endl;

	char raw_filename[LINE_SIZE];
	strcpy_s(raw_filename,"int_login_homegrown.txt");
	ifstream inClientFile (raw_filename,ios::in);
	if (!inClientFile)
	{
		cerr <<"anomalous file could not be opened" <<endl;
		exit(1);
	}

	ofstream outPrintFile1 ("testing_profile.txt", ios::out);
	if (!outPrintFile1)
	{
		cerr <<"int files names file could not be opened" <<endl;
		exit(1);
	}
	outPrintFile1 <<endl<<endl<<endl;
	outPrintFile1 <<"Start debugging"<<endl<<endl<<endl;
	int number_mismatches=0;
	int number_rows_in_testing_profile=0;
	int asyscall;
	int window_sized_array[WINDOW_SIZE];
	for (int i=0;i<WINDOW_SIZE;i++) window_sized_array[i] =-1;
	for (int abc=0;abc<WINDOW_SIZE-1;abc++)
```

```
		{
			if(inClientFile >> asyscall) window_sized_array[abc]=asyscall;
		}
	while (inClientFile >> asyscall)
	{
		window_sized_array[WINDOW_SIZE-1]=asyscall;
		number_rows_in_testing_profile++;
		outPrintFile1 <<"current testing input row:  ";
		for (int def=0;def<WINDOW_SIZE;def++)
			outPrintFile1 << window_sized_array[def] << " ";
		outPrintFile1 <<endl;


		if (table[window_sized_array[0]].isEmpty())
		{
			outPrintFile1 <<"the sequence: ";
			for (int x=0;x<WINDOW_SIZE;x++)
				outPrintFile1 << setw(5) << window_sized_array[x];
			outPrintFile1 <<"  does not have an entry in the training profile"<<endl;
			number_mismatches++;
		}else
		{
			SeqhashNode< int > *currentPtr = table[window_sized_array[0]].firstPtr;
			int seen=0;
			while(currentPtr != 0)
			{
				int same_size=0;
				for (int j=0;j<WINSIZE_MINUS_ONE;j++)
					if (currentPtr->sequence[j]== window_sized_array[j+1])
					same_size ++;
				if (same_size == WINSIZE_MINUS_ONE)seen++;
				currentPtr =  currentPtr->nextPtr;
			}

			if(seen==0)
			{
				outPrintFile1 <<"the sequence: ";
				for (int x=0;x<WINDOW_SIZE;x++)
					outPrintFile1 << setw(5) << window_sized_array[x];
				outPrintFile1 <<"  caused a mismatch"<<endl;
				number_mismatches ++;
			}
		}
		for (int ghi=0;ghi<WINDOW_SIZE-1;ghi++)
			window_sized_array[ghi]=window_sized_array[ghi+1];
		window_sized_array[WINDOW_SIZE-1]=-1;
	}
	time_t ending_time;
	time(&ending_time);
	outPrintFile2 << "Ending Time= "<<ending_time<<endl;
	outPrintFile2 << "Total testing Time= "<<ending_time-begining_time<<" seconds"<<endl<<endl;
	float mismatches_percentage= ((float)number_mismatches /
		(float)number_rows_in_testing_profile) * 100.00;
	outPrintFile2 <<"windows size "<<WINDOW_SIZE<<endl;
	outPrintFile2 <<"number of patterns in testing file  "<<number_rows_in_testing_profile<<endl;
```

```
		outPrintFile2 <<"Number of mismatches= "<<number_mismatches<<endl;
		outPrintFile2 <<"Percentage mismatches  = "<<mismatches_percentage<< " %"<<endl;
		SeqhashNode< int > *currentPtr;
		int number_of_patterns=0;
		for (int xx=0;xx<NUM_SYS_CALLS;xx++)
		{
			if (table[xx].isEmpty()){/*NOTHING*/}
			else
			{
				currentPtr = table[xx].firstPtr;
				while(currentPtr != 0)
				{
					number_of_patterns++;
					currentPtr =  currentPtr->nextPtr;
				}
			}
		}
		int any_table[WINSIZE_MINUS_ONE];
		outPrintFile2 <<"Number of patterns in normal database profile= "<<number_of_patterns<<endl;
		outPrintFile2 <<"Space cost of profile while at running time= "<<(sizeof (table) +
			( sizeof ( Seqhash<int>) * sizeof(any_table) *number_of_patterns) )<<" bytes"<<endl;
		outPrintFile2 <<"Space cost of profile while saved to disk= "
		<<sizeof(int)*WINDOW_SIZE*number_of_patterns<<" bytes"<<endl;
}
//---------------------------------------------------------------------------------------------------------------
//MAIN
//---------------------------------------------------------------------------------------------------------------
int main()
{
		int num_files=0;
		int total_number_profile_rows =0;
		char filename[LINE_SIZE];
		char filename_table[NUMBER_TRAINING_FILES][LINE_SIZE];
		strcpy_s(filename,"int_files_names.txt");
		ifstream inClientFile (filename,ios::in);
		if (!inClientFile)
		{
			cerr <<"File could not be opened" <<endl;
			exit(1);
		}
		char one_filename[LINE_SIZE];
		while (inClientFile >> one_filename)
		{
			strcpy_s (filename_table[num_files], one_filename);
			num_files++;
		}
		ofstream outPrintFile4 ("tracing.txt", ios::out);
			if (!outPrintFile4)
			{
				cerr <<"int files names file could not be opened" <<endl;
				exit(1);
			}
		int temp_sequence [ MAX_SEQ_SIZE];
		Seqhash<int> seq_hash_table[NUM_SYS_CALLS];
		for (int file_counter=0;file_counter<num_files;file_counter++)
```

```
{
        int sequence_profile_array[PROFILE_ROWS][WINDOW_SIZE];
        for (int i=0;i < PROFILE_ROWS; i++)
                for (int j=0;j<WINDOW_SIZE; j++)
                        sequence_profile_array[i][j] =-1;

        for (int i=0;i<MAX_SEQ_SIZE;i++) temp_sequence [ i] = -1;
        strcpy_s (one_filename,filename_table[file_counter]);
        ifstream inClientFile1 (one_filename,ios::in);
        if (!inClientFile1)
        {
                cerr <<"File could not be opened" <<endl;
                exit(1);
        }
        int seq_len=0;
        while (inClientFile1 >> temp_sequence[seq_len])seq_len++;
        int temp_row=0;
        for (int p=0;p<seq_len;p++)
        {
                for (int k=0; k<WINDOW_SIZE; k++)
                {
                        sequence_profile_array[temp_row][k]=temp_sequence[p+k];
                }
                temp_row++;
        }
        temp_row=temp_row-(WINDOW_SIZE-1);
        //ignoring the rows at the lower pyramid that

        outPrintFile4 <<"number of rows before removing redundant rows" <<
                temp_row <<endl;
        //---------------------------------------------------------------------------
        //removing redundant rows
        //---------------------------------------------------------------------------
        for (int i=0;i<temp_row-1;i++)
                for (int j=i+1;j<temp_row;j++)
                {
                        int alength=0;
                        for (int k=0;k<WINDOW_SIZE;k++)
                                if(sequence_profile_array[i][k]==
                                sequence_profile_array[j][k]) alength++;
                        if (alength==WINDOW_SIZE) sequence_profile_array[j][0]=-1;

                }
        int temp_sequence_profile_array[PROFILE_ROWS][WINDOW_SIZE];
        for (int i=0;i<PROFILE_ROWS;i++)
                for (int j=0;j<WINDOW_SIZE;j++)
                        temp_sequence_profile_array[i][j]=-1;
        int row_counter=0;
        for (int i=0;i<temp_row;i++)
        {
                if(sequence_profile_array[i][0]!=-1)
                {
                        for (int j=0;j<WINDOW_SIZE;j++)
                        {
```

```
                        temp_sequence_profile_array[row_counter][j]=
                        sequence_profile_array[i][j];
                }
                row_counter++;
        }

}
outPrintFile4 <<"number_profile_rows after removing redundant rows" <<
         row_counter <<endl;
for (int i = 0;i<row_counter;i++)
{
        for (int j=0;j<WINDOW_SIZE;j++)
                outPrintFile4 << temp_sequence_profile_array[i][j]<<" " ;
        outPrintFile4 <<endl;
}
//---------------------------------------------------------------------------
//save table information in sequence format
//---------------------------------------------------------------------------
SeqhashNode< int > *currentPtr;
int one_seq[WINDOW_SIZE];
int partial_seq[WINDOW_SIZE-1];

for (int i=0;i<row_counter;i++)
{
        for(int j=0;j<WINDOW_SIZE;j++) one_seq[j]=
        temp_sequence_profile_array[i][j];

        if (one_seq[0]!=-1)
        {
                for (int j=1;j<WINDOW_SIZE;j++) partial_seq[j-1] =one_seq[j];
                if (seq_hash_table[one_seq[0]].isEmpty())
                {
                        seq_hash_table[one_seq[0]].insertAtBack(partial_seq);
                }
                else
                {
                        currentPtr = seq_hash_table[one_seq[0]].firstPtr;
                        int found_match=0;
                        while(currentPtr != 0)
                        {
                                int xlen=0;
                                for(int x=0;x<WINDOW_SIZE-1;x++)
                                        if (currentPtr->sequence[x]==
                                        partial_seq[x])xlen++;
                                if (xlen==WINDOW_SIZE-1)found_match++;
                                currentPtr =  currentPtr->nextPtr;
                        }
                        if (found_match==0)
                        seq_hash_table[one_seq[0]].insertAtBack(partial_seq);
                }

        }

}
```

```
	}

	testing( seq_hash_table);
	return 0;
}
```

APPENDIX B

LOG FILE EXAMPLE OF NORMAL PATTERN DB FOR FILE “INT_509.TXT”

This section shows portions of the log file collected while generating fixed length detectors with different window sizes. The patterns (sub sequences) generated are for the file (int_509.txt) and mainly show the window size and then the nmber of rows before and after removing redundant entries. As shown the number of rows before removing redunadant data decresess by one as the window size increass by one. However the number of rows after removing redundant data (rows) does not follow any linear increase or decrease but depend on the data itself.

```
File name : int_509.txt
Window size = 3
number of rows before removing redundant rows = 362
number_profile_rows after removing redundant rows = 181
90 125 106
125 106 5
106 5 90
5 90 6
90 6 5
6 5 3
5 3 90
3 90 6
90 6 125
6 125 5
125 5 3
6 125 91
125 91 125
91 125 136
125 136 49
136 49 24
49 24 47
24 47 50
47 50 67
50 67 27
67 27 67
.
.
.
.
4 6 76
6 76 75
75 5 67
5 67 3
67 3 67
3 67 6
67 6 106
```

```
6 106 67
106 67 23
67 23 12
23 12 2
12 2 67
2 67 114
67 114 67
114 67 5
67 5 108
75 24 102
24 102 13
76 75 91
75 91 1
File name : int_509.txt
Window size = 4
number of rows before removing redundant rows361
number_profile_rows after removing redundant rows206
90 125 106 5
125 106 5 90
106 5 90 6
5 90 6 5
90 6 5 3
6 5 3 90
5 3 90 6
3 90 6 5
3 90 6 125
90 6 125 5
6 125 5 3
125 5 3 90
90 6 125 91
6 125 91 125
125 91 125 136
91 125 136 49
125 136 49 24
136 49 24 47
49 24 47 50
24 47 50 67
.
.
.
.
23 12 2 67
12 2 67 114
2 67 114 67
67 114 67 5
114 67 5 108
67 5 108 90
76 75 24 102
75 24 102 13
24 102 13 20
6 76 75 91
76 75 91 1
File name : int_509.txt
Window size = 5
number of rows before removing redundant rows = 360
```

```
number_profile_rows after removing redundant rows = 226
90 125 106 5 90
125 106 5 90 6
106 5 90 6 5
5 90 6 5 3
90 6 5 3 90
6 5 3 90 6
5 3 90 6 5
3 90 6 5 3
5 3 90 6 125
3 90 6 125 5
90 6 125 5 3
6 125 5 3 90
125 5 3 90 6
3 90 6 125 91
90 6 125 91 125
6 125 91 125 136
125 91 125 136 49
91 125 136 49 24
.
.
.
67 114 67 5 108
114 67 5 108 90
67 5 108 90 3
91 76 75 24 102
76 75 24 102 13
75 24 102 13 20
24 102 13 20 4
4 6 76 75 91
6 76 75 91 1
```

APPENDIX C

LOOKAHEAD-PAIRS METHOD BASED IDS SOURCE CODE

```
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//LOOKAHEAD PAIRS MEHTOD IDS
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//--------------------------------------------------------------------------------------------------------------
//MAIN
//--------------------------------------------------------------------------------------------------------------
void main()
{
		const int size = (WINDOW_SIZE-1)* NUM_SYS_CALLS*NUM_SYS_CALLS;
		std::bitset<size> b_array;
		int number_syscalls;
		char syscall_hashtable[NUM_SYS_CALLS][LINE_SIZE];
		number_syscalls = create_syscalls_hashtable(syscall_hashtable);
		char filename_table[NUMBER_TRAINING_FILES][LINE_SIZE];
		char filename[LINE_SIZE];
			strcpy_s(filename,"int_files_names.txt");
		ifstream inClientFile (filename,ios::in);
		if (!inClientFile)
		{
			cerr <<"File could not be opened" <<endl;
			exit(1);
		}
		char one_filename[LINE_SIZE];
		int counter=0;//number of files to be read
		while (inClientFile >> one_filename)
		{
			strcpy_s (filename_table[counter], one_filename);
			counter++;
		}
		Tree <int> TreeObject;
		List< int > listObject[NUMBER_TRAINING_FILES];
		for (int temp_counter =0;temp_counter<counter;temp_counter++)
		{
			TreeObject.insertAtRight(temp_counter+1);
			strcpy_s(filename, filename_table[temp_counter]);
			ifstream inClientFile1 (filename,ios::in);
			if (!inClientFile1)
			{
				cerr <<"File could not be opened" <<endl;
				exit(1);
			}
			int value;
		while (inClientFile1 >> value)
listObject[temp_counter].insertAtBack(value);
		TreeObject.lastPtr->downPtr = listObject[temp_counter].firstPtr;
}

ofstream outPrintFile ("tracing.txt", ios::out);
if (!outPrintFile)
{
		cerr <<"int files names file could not be opened" <<endl;
		exit(1);
}
int number_pairs;

TreeNode< int > *currentPtr = TreeObject.firstPtr;
ListNode<int> *listCurrentPtr;

while ( currentPtr!=0)
{
		int temp_row=0;
		int temp_sequence [MAX_SEQ_SIZE];

		int profile_array[PROFILE_ROWS][WINDOW_SIZE];
		for (int i=0;i < PROFILE_ROWS; i++)
				for (int j=0;j<WINDOW_SIZE; j++)
						profile_array[i][j] =-1;

		for (int i=0;i<MAX_SEQ_SIZE;i++) temp_sequence [ i] = -1;
		listCurrentPtr= currentPtr->downPtr;
		int seq_len=0;
		while (listCurrentPtr != 0)
		{
				temp_sequence[seq_len] = listCurrentPtr->data;
				listCurrentPtr =  listCurrentPtr->nextPtr;
				seq_len++;
		}
		for (int p=0;p<seq_len;p++)
		{
				for (int k=0; k<WINDOW_SIZE ; k++)
				{
						profile_array[temp_row][k]=temp_sequence[p+k];

				}
				temp_row++;
		}

		int number_profile_rows =0;
		for (int i=0; i<=temp_row;i++)
		{
				if (profile_array[i][WINDOW_SIZE-1] != -1)
				{
						number_profile_rows ++;
				}else for (int j=0;j<WINDOW_SIZE;j++)profile_array[i][j]=-1;
				//removing entries at the lower pyramid of profile
		}
		//adding the remaining upper pyramid data to the end of the array_profile
```

```
int pyramid_row_entries =  WINDOW_SIZE-2;
int y=1;
for(int i=number_profile_rows;i<(number_profile_rows+pyramid_row_entries);i++)
{
            for(int j=0;j<WINDOW_SIZE-y;j++)
            {
                        profile_array[i][j+y] =profile_array[0][j];
            }
            y++;
}

number_pairs=number_profile_rows*(WINDOW_SIZE-1);
number_pairs=number_pairs+((pyramid_row_entries*(WINDOW_SIZE-1))/2);
number_profile_rows = number_profile_rows + pyramid_row_entries;
for (int i=number_profile_rows;i<PROFILE_ROWS;i++)
            for(int j=0;j<WINDOW_SIZE;j++)
                        profile_array[i][j]=-1;
outPrintFile <<"number_profile_rows before removing redundant rows= " <<
            number_profile_rows <<endl;
outPrintFile <<"array content with pyramid data at end of array"<<endl;
for (int i = 0;i<number_profile_rows;i++)
{
            for (int j=0;j<WINDOW_SIZE;j++)
                        outPrintFile << profile_array[i][j]<<" " ;
            outPrintFile <<endl;
}
//---------------------------------------------------------------------------
//REMOVING REDUNDANT ENTRIES
//---------------------------------------------------------------------------

for (int i=0;i<number_profile_rows-1;i++)
            for (int j=i+1;j<number_profile_rows;j++)
            {
                        int alength=0;
                        for (int k=0;k<WINDOW_SIZE;k++)
                                    if(profile_array[i][k]==profile_array[j][k]) alength++;
                        if (alength==WINDOW_SIZE) profile_array[j][WINDOW_SIZE-1]=-
1;


            }
int temp_profile_array[PROFILE_ROWS][WINDOW_SIZE];
for (int i=0;i<PROFILE_ROWS;i++)
            for (int j=0;j<WINDOW_SIZE;j++)
                        temp_profile_array[i][j]=-1;
int row_counter=0;
for (int i=0;i<PROFILE_ROWS;i++)
{
            if(profile_array[i][WINDOW_SIZE-1]!=-1)
            {
                        for (int j=0;j<WINDOW_SIZE;j++)
                        {
                                    temp_profile_array[row_counter][j]=profile_array[i][j];
                        }
                        row_counter++;
            }
```

```
		}
		for (int i=0;i<PROFILE_ROWS;i++)
			for (int j=0;j<WINDOW_SIZE;j++)
				profile_array[i][j]=-1;

		for (int i=0;i<=row_counter;i++)
			for (int j=0;j<WINDOW_SIZE;j++)
				profile_array[i][j]=temp_profile_array[i][j];
		number_profile_rows=row_counter;
		outPrintFile <<"number_profile_rows after removing redundant rows= " <<
			number_profile_rows <<endl;

		for (int i = 0;i<number_profile_rows;i++)
		{
			for (int j=0;j<WINDOW_SIZE;j++)
				outPrintFile << profile_array[i][j]<<" " ;
			outPrintFile <<endl;
		}
		//---------------------------------------------------------------------------
		//Save table information in lookahead format
		//---------------------------------------------------------------------------

		outPrintFile<<"the locations that are set to 1:"<<endl;
		int array_num =0;
		for(int j=WINDOW_SIZE-2;j>=0;j--)
		{
			for(int i=0;i<number_profile_rows;i++)
			{

				int row=profile_array[i][WINDOW_SIZE-1];
				int col=profile_array[i][j];
				if((row==-1)||(col==-1))
				{
					//NOTHING
				}else
				{
					outPrintFile <<"<"<<row<<","<<col<<">"<<" at plane: "<<
					array_num<<endl;
					b_array.set(  ((row-1)*NUM_SYS_CALLS)
					+col+
					(NUM_SYS_CALLS*NUM_SYS_CALLS*array_num)  );
				}
			}
			array_num++;
		}
		currentPtr = currentPtr->rightPtr;
	}//while (currentPtr!=0)

	//------------------------------------------------------------------------------------------
	//START TESTING
	//------------------------------------------------------------------------------------------
	time_t begining_time;
	time(&begining_time);
	outPrintFile << "Begining Time= "<<begining_time<<endl;
	char filename1[LINE_SIZE];
```

```
strcpy_s(filename1,"int_ps_homegrown.txt");
ifstream inClientFile1(filename1,ios::in);
if (!inClientFile1)
{
        cerr <<"anomalous file could not be opened" <<endl;
        exit(1);
}
int number_rows_in_testing_profile=0;
int number_of_pairs_in_testing_profile=0;
int value;
int window_sized_array[WINDOW_SIZE];
for (int i=0;i<WINDOW_SIZE;i++) window_sized_array[i] =-1;
for (int abc=0;abc<WINDOW_SIZE-1;abc++)
        {
                if(inClientFile >> value) window_sized_array[abc]=value;
        }
double percentage;
int mismatches=0;
int row, col;
while (inClientFile1 >> value)
{
        window_sized_array[WINDOW_SIZE-1]=value;
        number_rows_in_testing_profile++;
        outPrintFile <<"current testing input row:  ";
        for (int def=0;def<WINDOW_SIZE;def++) outPrintFile <<
                window_sized_array[def] << " ";
        outPrintFile <<endl;

        for (int i=0;i<WINDOW_SIZE-1;i++)
        {
                row=window_sized_array[WINDOW_SIZE-1];
                col=window_sized_array[WINDOW_SIZE-2-i];
                if ((row==-1)||(col==-1))
                {
                        //NOTHING
                }else
                {
                        number_of_pairs_in_testing_profile++;
                        if (b_array.test(  ((row-1)*NUM_SYS_CALLS) +col +
                        (NUM_SYS_CALLS*NUM_SYS_CALLS*i)  )==0)
                        {
                                mismatches ++;
                                outPrintFile <<"mismatch pair: <"<<row<<","<<col<<
                                "> at plain: "<<i<<endl;
                        }
                }
        }
        for (int ghi=0;ghi<WINDOW_SIZE-1;ghi++)
                window_sized_array[ghi]=window_sized_array[ghi+1];
        window_sized_array[WINDOW_SIZE-1]=-1;
}
//----------------------------------------------------------------------------------------------------------
time_t ending_time;
time(&ending_time);
outPrintFile << "Number of rows in testing profile= "<<
```

```
                number_rows_in_testing_profile-1<<endl;
        outPrintFile << "Number of pairs in testing profile= "<<
                number_of_pairs_in_testing_profile<<endl;
        outPrintFile << "Ending Time= "<<ending_time<<endl;
        outPrintFile << "Total testing Time= "<<ending_time-begining_time<<" seconds"<<endl<<endl;
        percentage = ((double) mismatches / number_of_pairs_in_testing_profile)* 100.0;
        outPrintFile << "Number of lookahead mismatches= "<<mismatches<<endl;
        outPrintFile << "Percentage of mismatches (anomaly sensitivity)= "<<percentage<<" %"<<endl;
        outPrintFile <<" Maximum number of lookahead-pairs SETS= "<<WINDOW_SIZE<<
                " sets"<<endl;
        outPrintFile <<" Minimum number of lookahead-pairs SETS= "<<WINDOW_SIZE-2<<
                " sets"<<endl;
        outPrintFile <<" Number of lookahaead pairs= "<<number_pairs<<endl;
        outPrintFile <<"Number of sets (planes)= "<<WINDOW_SIZE-1<<
                "planes. Each is a 256 x 256 bit array and NUM_SYS_CALLS=256"<<endl;
        outPrintFile <<"Space cost of profile while at running time= "<<b_array.size() <<
                " bits. = "<<(b_array.size()/8)<<" bytes."<< endl;
        outPrintFile <<"Space cost of profile while saved to disk= "<<sizeof(int)*number_pairs<<
                " bytes"<<endl;
        cout <<"exiting"<<endl;
}
```

APPENDIX D

VARIABLE LENGTH DETECTORS WITH OVERLAP RELATIONSHIP METHOD BASED IDS SAMPLE CODE

```
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//VARIABLE LENGTH DETECTPRS BASED IDS
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

//---------------------------------------------------------------------------
//TESTING
//---------------------------------------------------------------------------

void testing()
{
        Seqhash<int> table[NUMBER_SYSCALLS]; //hash table of normal pattern database
        ofstream outPrintFile22 ("tracing_variable.txt", ios::out);
        if (!outPrintFile22)
        {
                cerr <<"int files names file could not be opened" <<endl;
                exit(1);
        }
        time_t begining_time;
        time(&begining_time);
        outPrintFile22 << "Begining Time= "<<begining_time<<endl;
        int number_mismatches=0;
        int number_systecalls_read=0;
        strcpy_s(filename,"int_login_homegrown.txt");
        ifstream inClientFile (filename,ios::in);
        if (!inClientFile)
        {
                cerr <<"anomalous file could not be opened" <<endl;
                exit(1);
        }
        int going_to_new_pattern;
        int value;
        int not_seen_syscall=0;
        int prev_value;
        if (inClientFile >> value)
        {
        do
        {
        number_systecalls_read++;
```

```
int counter =0;
int type=0;
prev_value = value;
if (table[value].isEmpty())
{
        number_mismatches++;
        not_seen_syscall++;
}else
{
        SeqhashNode< int > *currentPtr = table[value].firstPtr;
        if (inClientFile >> value)
        {

                int s=1;
         do
         {
                number_systecalls_read++;
                going_to_new_pattern =0;
                prev_value = value;
                while ((currentPtr->nextPtr != 0) &&(currentPtr->sequence[s]!=value))
                {
                        currentPtr =  currentPtr->nextPtr;
                }
                if ((currentPtr == 0) || (currentPtr->sequence[s]!=value))
                {
                        number_mismatches ++;
                        counter++;
                }
                else if (currentPtr->sequence[s]== value)
                {
                int j=s+1;
                int count2=0;
                int there_is_systemcalls=1;
                while ((currentPtr != 0) &&(currentPtr->sequence[j]!= -1)
                        &&(there_is_systemcalls ==1))
                {
                there_is_systemcalls=0;
                if (inClientFile >> value)
                {
                number_systecalls_read++;
                there_is_systemcalls =1;
                prev_value = value;

                //the following to handle overlaping patterns
                if ((currentPtr->sequence[j]!= value)
                                &&(table[value].isEmpty())&& (count2>2))
                {
                        count2++;
                        type=1;
                }
                if ((currentPtr->sequence[j]!= value)
                        &&(!table[value].isEmpty())&& (count2>2))
                {
                        count2++;
                        type=2;
                }
                if ((currentPtr->sequence[j]!= value)
                        &&(!table[prev_value].isEmpty())&& (count2>2))
                {
                        count2++;
                        type=3;
                }
                if (currentPtr->sequence[j]!= value)
                {
                        number_mismatches ++;
                }
                if (count2>2)
                switch (type)
                {
                case 1:

                        number_mismatches ++;
                        not_seen_syscall++;
                        break;
                case 2:
                        currentPtr = table[value].firstPtr;
                        j=0;
                        break;
                case 3:
                        currentPtr = table[prev_value].firstPtr;
                        j=1;
                        break;
                }
                j++;
                }//if
                }//while
                if (currentPtr->sequence[j]== -1) going_to_new_pattern = 1;
                }//elseif
                s++;
        }while((going_to_new_pattern ==0)&&(counter <3)&&(inClientFile >> value));
        }//if
}//else
}while (inClientFile >> value);
}//if
time_t ending_time;
time(&ending_time);
outPrintFile22 << "Ending Time= "<<ending_time<<endl;
outPrintFile22 << "Total testing Time= "<<ending_time-begining_time<<
        " seconds"<<endl<<endl;
outPrintFile22 << "number of mismatches: "<<number_mismatches<<endl;
outPrintFile22 <<"size of testing file: "<<number_systecalls_read<<" system calls"<<endl;
float mismatches_percentage= ((float)number_mismatches / (float)number_systecalls_read) *
         100.00;
outPrintFile22 <<"Percentage of sequence mismatches= "<<mismatches_percentage<<
        " %"<<endl;
int number_of_patterns=0;
int max_pattern_length=0;
int min_pattern_length=100;
int number_of_syscalls_in_patterns=0; //number of system calls in all patterns in training database
for (int xx=1;xx<NUMBER_SYSCALLS;xx++)
```

```
		{
			if (table[xx].isEmpty())
			{
				//NOTHING
			}else
			{
				SeqhashNode< int > *currentPtr = table[xx].firstPtr;
				while(currentPtr != 0)
				{
				number_of_patterns++;
				int yy=0;
				while(currentPtr->sequence[yy] != -1)
				{
					number_of_syscalls_in_patterns++;
					yy++;
				}
				if (yy>max_pattern_length) max_pattern_length = yy;
				if (yy<min_pattern_length) min_pattern_length = yy;

				currentPtr =  currentPtr->nextPtr;
				}
			}
		}
		int for_testing[PINSTANCE_SIZE];
		outPrintFile22 << "number of patterns in training database: "<<number_of_patterns<<endl;
		outPrintFile22 << "Max pattern length: "<<max_pattern_length<<endl;
		outPrintFile22 << "Min pattern lenfth: "<<min_pattern_length<<endl;
		outPrintFile22 << "Average pattern length: "
		<<((max_pattern_length+min_pattern_length)/2)<<endl;
		outPrintFile22 <<"size of normal file: "<<number_of_syscalls_in_patterns<<
		" system calls"<<endl;
		outPrintFile22 <<"Space cost of profile while at running time= "<<(sizeof (table) +
			(sizeof ( Seqhash<int>) * sizeof(for_testing)* number_of_patterns) )<<" bytes"<<endl;
		outPrintFile22 <<"Space cost of profile while saved to disk= "
		<<sizeof(int)*number_of_syscalls_in_patterns<<" bytes"<<endl;
}
//---------------------------------------------------------------------------
//READING TRAINING FILES AND GENERATING NORMAL PATTERN DATABASE WITH
POSITIVE DETECTORS
//---------------------------------------------------------------------------

int pattern_candidates_array[PINSTANCE_SIZE][PINSTANCE_SIZE];//first row contain the size of the
pattern starting from row1
fill_array_with_negone(pattern_candidates_array);
char filename_table[NUMBER_SYSCALLS][LINE_SIZE];

Seqhash<int> adjacency_list[NUMBER_SYSCALLS];
int number_syscalls;

int max_seq_size= 0;
char filename[LINE_SIZE];
strcpy_s(filename,"all_training_files2.txt");
ifstream inClientFile (filename,ios::in);
if (!inClientFile)
{
        cerr <<"File could not be opened" <<endl;
        exit(1);
}
char one_filename[LINE_SIZE];
int counter=0;//number of files to be read
while (inClientFile >> one_filename)
{
        strcpy_s (filename_table[counter], one_filename);
        counter++;
}

Tree <int> TreeObject;
Pinstance <int> PinstanceObject;
List< int > listObject[NUMBER_TRAINING_FILES];
int seen[NUMBER_SYSCALLS];
for (int k=0;k<NUMBER_SYSCALLS;k++) seen[k]=-1;
int number_tobe_processed=0;
for (int temp_counter =0;temp_counter<counter;temp_counter++)
{
        TreeObject.insertAtRight(temp_counter+1);
        strcpy_s(filename, filename_table[temp_counter]);
        ifstream inClientFile1 (filename,ios::in);
        if (!inClientFile1)
        {
                cerr <<"File could not be opened" <<endl;
                exit(1);
        }
        int value;
        while (inClientFile1 >> value)
        {
                seen[value]=0;
                listObject[temp_counter].insertAtBack(value);
        }
        TreeObject.lastPtr->downPtr = listObject[temp_counter].firstPtr;
}
for (int k=0;k<NUMBER_SYSCALLS;k++)
        if(seen[k]==0)number_tobe_processed++;
TreeNode<int> *headTempPtr,*headTempPtrOuter;
ListNode<int> *tempPtr;
headTempPtrOuter = TreeObject.firstPtr;
int instance_counter=0;
int temp_pinstance[PINSTANCE_SIZE][PINSTANCE_SIZE];

int temp_e = TreeObject.firstPtr->downPtr->data;//starting from first element as the
                                                //never seen before system call
seen[temp_e]=1;
for(int temp_counter=1;temp_counter <= counter; temp_counter++)
{
        tempPtr=headTempPtrOuter->downPtr;
        do
        {
                if (tempPtr->data == temp_e)
                {
                        instance_counter++;
                        tempPtr->instance_value=instance_counter;
                }
                tempPtr=tempPtr->nextPtr;
        }while ((tempPtr !=0)&&(headTempPtr != 0));
        for (int i=0;i<PINSTANCE_SIZE; i++)
                for (int j=0;j<PINSTANCE_SIZE; j++)
                {
                        temp_pinstance[i][j]=-1;
                }
        headTempPtr = TreeObject.firstPtr;
        for(int temp_counter=1;temp_counter <=instance_counter; temp_counter++)
        {
                tempPtr=headTempPtr->downPtr;
                while ((tempPtr->data != temp_e)&&(tempPtr != 0))
                {
                        tempPtr=tempPtr->nextPtr;
                }
                while((tempPtr !=0)&&(headTempPtr != 0))
                {
                        if (tempPtr !=0)
                        {
                        if (tempPtr->data == temp_e)
                        {
                        int temp_row =0;
                        int temp_col =tempPtr->instance_value;
                        temp_pinstance[temp_row][temp_col] = tempPtr->data;
                        tempPtr=tempPtr->nextPtr;
                        temp_row++;
                        while ((tempPtr != 0)&&(tempPtr->data != temp_e))
                        {
                        if (tempPtr != 0)
                        {
                                temp_pinstance[temp_row][temp_col] = tempPtr->data;
                                tempPtr=tempPtr->nextPtr;
                                temp_row ++;
                        }
                        if (tempPtr == 0)
                        {
                                temp_pinstance[temp_row][temp_col] = -1;
                                temp_row ++;
                        }
                        }
                        if (temp_row > max_seq_size) max_seq_size = temp_row;
                        }
                        }
                }

        }
        int temp_array [PINSTANCE_SIZE][PINSTANCE_SIZE];
        fill_array_with_negone(temp_array);
        int pinstance_value=0;
        int num_maximal_pattern_candidates =0;
        for (int j=0;j<PINSTANCE_SIZE;j++) temp_array[0][j]=temp_pinstance[0][j];
        Pinstance <int> pinstanceObject;
        pinstanceObject.insertRootNode (pinstance_value, temp_array);
        pinstance_value++;
```

```
        int i=1;
        int number_columns = instance_counter;
        int last_temp_array [PINSTANCE_SIZE][PINSTANCE_SIZE];
        int temp_temp_pinstance[PINSTANCE_SIZE][PINSTANCE_SIZE];
        for(int z =0;z<PINSTANCE_SIZE;z++)
                for(int x=0;x<PINSTANCE_SIZE;x++)
                        temp_temp_pinstance[z][x]=temp_pinstance[z][x];

        //--------------------------------------------------------------------------
        //START TYPE CHECKING
        //--------------------------------------------------------------------------
        int num_repeated[NUMBER_SYSCALLS];
        Type5List< int > type5ListObject;
        int num_subsections =1;
        int temp_last_cols =1;
        while ((num_subsections > 0)&&(number_columns>1))
        {
                int just_done_type5=0;
                int type1Found =0;
                int type2Found =0;
                int type3Found=0;
                int type4Found=0; //there is a type 1 and a type 2
                int type5Found=0; //type 2 is divided into two or more sections
                for(int j=0;j<PINSTANCE_SIZE;j++)
                        temp_array[i][j]=temp_temp_pinstance[i][j];
                fill_array_with_negone(last_temp_array);
                int temp_holder = temp_array[i][1];
                for (int c=1;c<=number_columns;c++)
                {
                        if (temp_array[i][c] == -1) type1Found =1;
                        if ((temp_holder != temp_array[i][c]) &&(temp_array[i][c]!=-1)) type2Found
=1;
                }
                if ((type1Found ==1) && (type2Found ==1) )
                {
                        type4Found =1;
                }
                if (type2Found ==1)
                {
                        for (int countKK=0;countKK<NUMBER_SYSCALLS;countKK++)
                                num_repeated[countKK]=0;
                        for (int countKK=1;countKK<=number_columns;countKK++)
                                if (temp_array[i][countKK]!=-1)
                                num_repeated[temp_array[i][countKK]]++;
                        int more_than_1=0;
                        for (int countKK=1;countKK<NUMBER_SYSCALLS;countKK++)
                                if ((num_repeated[countKK])>1) more_than_1 ++;
                        if (more_than_1 >1) type5Found =1;
                }
                if ((type1Found == 0)&&(type2Found ==0)) type3Found =1;
                else type3Found =0;

                if ((type1Found ==1)&&(type2Found==1)&&(number_columns <=2))
                {
                        type1Found =0;
```

```
		type2Found =0;
		type3Found=0;
		type4Found=0;
		type5Found=0;
		num_subsections=0;
		number_columns=0;
	}
	//-------------------------------TYPE4---------------------------------------
	if  (type4Found == 1)
	{
		int temp_right_col=1;
		int temp_left_col=1;
		for (int c=1;c<=number_columns;c++)
		{
			if (temp_array[i][c] == -1)
			{
				for(int m =0;m <i;m ++)
				last_temp_array[m][temp_left_col]=temp_array[m][c];
				temp_left_col ++;
				for (int abc=0;abc<PINSTANCE_SIZE;abc++)
				temp_temp_pinstance[abc][c]=-1;
				for (int abc=0;abc<PINSTANCE_SIZE;abc++)
				temp_array[abc][c]=-1;
			}
		}
		pinstanceObject.insertNode(pinstance_value,last_temp_array,0);//left
		pinstance_value++;
		num_maximal_pattern_candidates ++;
		for (int m=1;m<=number_columns;m++)
		{
		if (temp_array[i][m]==-1)
		{
		for (int n=m;n<=number_columns-1;n++)
		for (int p=0;p<PINSTANCE_SIZE;p++)
		{
			temp_array[p][n]=temp_array[p][n+1];
			temp_temp_pinstance[p][n]=temp_temp_pinstance[p][n+1];
		}
		number_columns--;
		}
		}
		for (int m=1;m<=number_columns;m++)
		{
		if (temp_array[i][m]!=-1) if (num_repeated[temp_array[i][m]]==1)
		{
		for (int n=m;n<=number_columns-1;n++)
		for (int p=0;p<PINSTANCE_SIZE;p++)
		{
			temp_array[p][n]=temp_array[p][n+1];
			temp_temp_pinstance[p][n]=temp_temp_pinstance[p][n+1];
		}
		number_columns--;
		}
		}
		pinstanceObject.insertNode(pinstance_value,temp_array,1);//right
		pinstance_value++;
		i++;
	//-------------------------------TYPE1---------------------------------------
	}else if  (type1Found == 1)
	{
		int temp_right_col=1;
		int temp_left_col=1;
		for (int c=1;c<=number_columns;c++)
		{
		if (temp_array[i][c] == -1)
		{
		for(int m =0;m <i;m ++)
			last_temp_array[m][temp_left_col]=temp_array[m][c];
		temp_left_col ++;
		for (int abc=0;abc<PINSTANCE_SIZE;abc++)temp_temp_pinstance[abc][c]
			=-1;
		for (int abc=0;abc<PINSTANCE_SIZE;abc++)temp_array[abc][c]=-1;
		}
		}
		pinstanceObject.insertNode(pinstance_value,last_temp_array,0);//left
		pinstance_value++;
		num_maximal_pattern_candidates ++;
		for (int m=1;m<=number_columns;m++)
		{
		if (temp_array[i][m]==-1)
		{
		for (int n=m;n<=number_columns-1;n++)
		for (int p=0;p<PINSTANCE_SIZE;p++)
		{
			temp_array[p][n]=temp_array[p][n+1];
			temp_temp_pinstance[p][n]=temp_temp_pinstance[p][n+1];
		}
		number_columns--;
		}
		}
		pinstanceObject.insertNode(pinstance_value,temp_array,1);//right
		pinstance_value++;
		i++;
	//-------------------------------TYPE2 AND TYPE5---------------------------------------
	}else if (type2Found == 1)
	{
		for (int m=1;m<=number_columns;m++)
		{
		if (temp_array[i][m]!=-1)if (num_repeated[temp_array[i][m]]==1)
		{
		for (int n=m;n<=number_columns-1;n++)
		for (int p=0;p<PINSTANCE_SIZE;p++)
		{
			temp_array[p][n]=temp_array[p][n+1];
			temp_temp_pinstance[p][n]=temp_temp_pinstance[p][n+1];
		}
		number_columns--;
		}
		}
		if (type5Found == 1)
		{
		num_subsections--;
		int getting_size;
		int maximum_size=0;
		for (int hh=1;hh<NUMBER_SYSCALLS;hh++)
			if (num_repeated[hh] >1)
			{
			type5ListObject.insertAtBack();
			int counter3=1;
			for (int counter1=0;counter1<PINSTANCE_SIZE;counter1++)
			{
			if (temp_temp_pinstance[i][counter1]==hh)
			{
			getting_size=0;
			for (int counter2=0;counter2<PINSTANCE_SIZE;counter2++)
			{
				if (temp_temp_pinstance[counter2][counter1]!=-1)
				getting_size++;
				type5ListObject.lastPtr->data[counter2][counter3]
				=temp_temp_pinstance[counter2][counter1];
				type5ListObject.lastPtr->starting_row=i;
			}
			if (getting_size > maximum_size) maximum_size = getting_size;
			counter3++;
			}
			}
			type5ListObject.lastPtr->max_seq_len=maximum_size+1;
			type5ListObject.lastPtr->num_columns=counter3-1;
			num_subsections++;
			}
		//filling temp_temp_pinstance with -1
		for (int yy=0;yy< PINSTANCE_SIZE;yy++)
				for (int xx=0;xx<PINSTANCE_SIZE;xx++)
					temp_temp_pinstance[yy][xx]=-1;
		for (int yy=0;yy< PINSTANCE_SIZE;yy++)
				for (int xx=0;xx<PINSTANCE_SIZE;xx++)
					temp_array[yy][xx]=-1;

		just_done_type5=1;
		}
		i++;
	//--------------------------------TYPE3----------------------------------------
	}else if (type3Found == 1)
	{
		i++;
		int number_of_minus_ones=0;
		for (int x=1;x<=number_columns;x++)
		{
			temp_array[i][x] =  temp_temp_pinstance[i][x];
			if(temp_array[i][x]==-1) number_of_minus_ones++;
		}

		if (number_of_minus_ones>=number_columns)
		{
			num_subsections--;
```

```
                }else pinstanceObject.modifyNode(temp_array);

        }
        //--------------------------------END ALL TYPE CHECKING-----------------------------
        if (just_done_type5==1)
        {
                if ( type5ListObject.firstPtr != 0 )
                {
                        for (int yy=0;yy< PINSTANCE_SIZE;yy++)
                                for (int xx=0;xx<PINSTANCE_SIZE;xx++)
                                        temp_temp_pinstance[yy][xx]=-1;
                        for (int yy=0;yy< PINSTANCE_SIZE;yy++)
                                for (int xx=0;xx<PINSTANCE_SIZE;xx++)
                                        temp_temp_pinstance[yy][xx]=
                                        type5ListObject.firstPtr->data[yy][xx];
                        i=type5ListObject.firstPtr->starting_row;
                        for (int xx=0;xx<i;xx++)
                                for (int yy=0;yy<PINSTANCE_SIZE;yy++)
                                        temp_array[xx][yy]=temp_temp_pinstance[xx][yy];
                        max_seq_size=type5ListObject.firstPtr->max_seq_len;
                        number_columns=type5ListObject.firstPtr->num_columns;
                        type5ListObject.removeFromFront();
                        num_subsections--;
                }
        }
}//End of while

//-----------------------------------------------------------------------------------
filling_pattern_candidates_array(pinstanceObject.rootPtr, pattern_candidates_array,0);

int aa_pattern_candidate_array[PINSTANCE_SIZE][PINSTANCE_SIZE];
for (int xx=0;xx<PINSTANCE_SIZE;xx++)
        for (int yy=0;yy<PINSTANCE_SIZE;yy++)
                aa_pattern_candidate_array[xx][yy]=-1;
int temp_yy=1;
for (int yy=0;yy<PINSTANCE_SIZE;yy++)
        if(pattern_candidates_array[1][yy] != -1)
        {
                for (int xx=0;xx<PINSTANCE_SIZE;xx++)

aa_pattern_candidate_array[xx][temp_yy]=pattern_candidates_array[xx][yy];
                temp_yy ++;
        }
for (int xx=0;xx<PINSTANCE_SIZE;xx++)
        for (int yy=0;yy<PINSTANCE_SIZE;yy++)
                pattern_candidates_array[xx][yy]=aa_pattern_candidate_array[xx][yy];

num_maximal_pattern_candidates =0;
for (int x=0;x<PINSTANCE_SIZE;x++)
        if(pattern_candidates_array[1][x]!=-1)
        {
                num_maximal_pattern_candidates++;
                int seq_size=0;
                for (int y=1;y<PINSTANCE_SIZE;y++)
```

```
                        if (pattern_candidates_array[y][x]!=-1)seq_size++;
                pattern_candidates_array[0][x]=seq_size;
        }
//removing one element sequences
for (int i=1;i<=num_maximal_pattern_candidates;i++)
{
        if (pattern_candidates_array[0][i] < 2)
                for (int j=0;j<PINSTANCE_SIZE; j++) pattern_candidates_array[j][i]=-1;
}

for (int xx=0;xx<PINSTANCE_SIZE;xx++)
        for (int yy=0;yy<PINSTANCE_SIZE;yy++)
                aa_pattern_candidate_array[xx][yy]=-1;
 temp_yy=1;

        for (int yy=0;yy<PINSTANCE_SIZE;yy++)
                if(pattern_candidates_array[1][yy] != -1)
                {
                        for (int xx=0;xx<PINSTANCE_SIZE;xx++)

aa_pattern_candidate_array[xx][temp_yy]=pattern_candidates_array[xx][yy];
                        temp_yy ++;
                }
for (int xx=0;xx<PINSTANCE_SIZE;xx++)
        for (int yy=0;yy<PINSTANCE_SIZE;yy++)
                pattern_candidates_array[xx][yy]=aa_pattern_candidate_array[xx][yy];

char syscall_hashtable[NUMBER_SYSCALLS][LINE_SIZE];
number_syscalls = create_syscalls_hashtable(syscall_hashtable);
int one_seq[PINSTANCE_SIZE];
for (int j=0;j<PINSTANCE_SIZE;j++) one_seq[j]=-1;
num_maximal_pattern_candidates=temp_yy-1;
for (int i=1;i<=num_maximal_pattern_candidates;i++)
{
        if (pattern_candidates_array[0][i] != -1)
        {
        for (int j=0;j<PINSTANCE_SIZE-1;j++)
        {
                one_seq[j]=pattern_candidates_array[j+1][i];
        }
        int seq_length =0;
        while(one_seq[seq_length]!=-1)seq_length++;
        seq_length++;
        int pattern_exist=0;

                if (!adjacency_list[one_seq[0]].isEmpty())
                {
                int a=0;
                SeqhashNode< int > *currentPointer = adjacency_list[one_seq[0]].firstPtr;
                int CCC=0;
                for (int ee=0;ee<PINSTANCE_SIZE;ee++) if (one_seq[ee]!=-1)CCC++;
                while(currentPointer != 0)
                {
                int CCC2=0;
                for(int ee=0;ee<PINSTANCE_SIZE;ee++)
```

```
                    if ((currentPointer->sequence[ee]!=-1)&&
                            (currentPointer->sequence[ee]==one_seq[ee]))
                            CCC2++;
            if(CCC=CCC2) pattern_exist=1;

            currentPointer =  currentPointer->nextPtr;
            }
            }

    if (pattern_exist==0)
    {
            int sizeC=0;
            for (int CC=0;CC<PINSTANCE_SIZE;CC++)
                    if(one_seq[CC] !=-1)sizeC++;
            if (sizeC>1)
            {
            adjacency_list[one_seq[0]].insertAtBack(one_seq);
            TreeNode<int> *headPtrB;
            ListNode<int> *tempPtrB;
            ListNode<int> *temptempPtrB;
            headPtrB=TreeObject.firstPtr;
            while (headPtrB !=0)
            {
            tempPtrB=headPtrB->downPtr;
            while (tempPtrB !=0)
            {
            if(tempPtrB->data == one_seq[0])
            {
            temptempPtrB=tempPtrB;
            int temp_seq_size =0;
            int same=1;
            while ((temptempPtrB !=0)&&(temp_seq_size<seq_length-1))
            {
                    if (temptempPtrB->data != one_seq[temp_seq_size]) same =0;
                    temp_seq_size++;
                    temptempPtrB=temptempPtrB->nextPtr;
            }
            if (temp_seq_size != seq_length-1)
            {
                    same =0;
            }
            if (same ==1)
            {
            for (int p=0;p<seq_length-1;p++)
            {
            if (tempPtrB != 0)
            {
                    tempPtrB->belong_to_a_pattern =1;
                    tempPtrB= tempPtrB->nextPtr;
            }
            }
            }
            }
            if (tempPtrB != 0) tempPtrB=tempPtrB->nextPtr;
            }
```

```
                                                headPtrB=headPtrB->rightPtr;
                                                }
                                                }
                                    }
                                    }
                        }




//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//COLLECTING STAND ALONE SEQUENCES
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

int sequenceS[PINSTANCE_SIZE];
TreeNode<int> *headPtrS;
ListNode<int> *tempPtrS;
headPtrS=TreeObject.firstPtr;
int seq_exsist;
while (headPtrS !=0)
{
            tempPtrS=headPtrS->downPtr;
            while (tempPtrS !=0)
            {
                        while ((tempPtrS !=0)&&(tempPtrS->belong_to_a_pattern==1))
                                    tempPtrS=tempPtrS->nextPtr;
                        int countS=0;
                        for (int j=0;j<PINSTANCE_SIZE;j++) sequenceS[j]=-1;
                        while ((tempPtrS !=0)&&(tempPtrS->belong_to_a_pattern==0))
                        {
                                    sequenceS[countS]= tempPtrS->data;
                                    countS++;
                                    tempPtrS=tempPtrS->nextPtr;
                        }
                        if (sequenceS[0]!=-1)
                        {
                        seq_exsist =0;
                        if (!adjacency_list[sequenceS[0]].isEmpty())
                        {
                        SeqhashNode< int > *currentPointer = adjacency_list[sequenceS[0]].firstPtr;
                        while((currentPointer != 0)&&(seq_exsist==0))
                        {

                        int seqC2=0;
                        int seqC3=0;
                        for (int vv=0;vv<PINSTANCE_SIZE;vv++)
                        {
                                    if (sequenceS[vv]!=-1)seqC2++;
                                    if ((currentPointer->sequence[vv] ==sequenceS[vv])&&
                                                (sequenceS[vv]!=-1))seqC3++;
                        }
                        if(seqC2 == seqC3)
                        {
                                    seq_exsist =1;
                        }
                        if(currentPointer !=0) currentPointer =  currentPointer->nextPtr;
```

```
				}
				}
				if (seq_exsist==0)
				{
					int sizeC=0;
					for (int CC=0;CC<PINSTANCE_SIZE;CC++)
					if(sequenceS[CC]!=-1) sizeC++;
					if (sizeC>1)adjacency_list[sequenceS[0]].insertAtBack(sequenceS);
				}
				}
			}
			if(headPtrS!=0 )headPtrS=headPtrS->rightPtr;
		}
```

APPENDIX E

POSITIVE DETECTOR GENERATION

```
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//NEGATIVE DETECTOR GENERATION
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

        //-----------------------------------------------------------------------------------------
        // GENERATING AND SAVING NEGATIVE DETECTORS
        //-----------------------------------------------------------------------------------------
        ofstream outPrintFile777 ("naiveR0.txt", ios::out);
        if (!outPrintFile777)
        {
                cerr <<" file could not be opened" <<endl;
                exit(1);
        }
        int max_number_detectors =160; //CHANGE IT TO THE SIZE OF DECTORS NEEDED
        int R0_naive_array[R0_NAIVE_ROWS][WINDOW_SIZE];
        int R0_array[R0_ROWS][WINDOW_SIZE];
        srand(time(0));
        for (int abc=0;abc<R0_ROWS;abc++)
        {
                for (int def=0;def<WINDOW_SIZE;def++)
                {
                        R0_naive_array[abc][def]=1+rand()% number_syscalls;
                        outPrintFile777 <<setw(5)<<R0_naive_array[abc][def];
                }
                outPrintFile777 <<endl;
        }
        int abc=0;
        int number_inserted=0;
        for (int row_counter=0;row_counter<max_number_detectors;row_counter++)
        {
                int found_similar=0;
                for (int ghi=0;ghi<number_profile_rows;ghi++)
                {
                        if (R0_naive_array[abc][0]==profile_array[ghi][0])
                        {
                                int same_count=0;
                                for (int def=1;def<WINDOW_SIZE;def++)

                                if(R0_naive_array[abc][def]==
                                        profile_array[ghi][def])same_count++;
                                if (same_count >= WINDOW_SIZE-1)found_similar=1;
                        }
```

```
		}
		if(found_similar ==0)
		{
			int exists=0;
			for (int ghi=0;ghi<number_inserted;ghi++)
			{
				if (R0_naive_array[abc][0]==R0_array[ghi][0])
				{
					int same_count=0;
					for (int def=1;def<WINDOW_SIZE;def++)

	if(R0_naive_array[abc][def]==R0_array[ghi][def])same_count++;
					if (same_count >= WINDOW_SIZE-1)exists=1;
				}
			}
			if(exists ==1)
			{

				row_counter=row_counter-1;

			}else
			{
				number_inserted++;
				for (int def=0;def<WINDOW_SIZE;def++)
					R0_array[row_counter][def]=R0_naive_array[abc][def];
			}
		}
		abc++;

	}

	ofstream outPrintFile888 ("R0.txt", ios::out);
	if (!outPrintFile888)
	{
		cerr <<" file could not be opened" <<endl;
		exit(1);
	}
	for (int abc=0;abc<max_number_detectors;abc++)
	{
		for (int def=0;def<WINDOW_SIZE;def++)
		{

			outPrintFile888 <<setw(5)<<R0_array[abc][def];
		}
		outPrintFile888 <<endl;
	}
	std::bitset<size> c_array;
	int array_num =0;
	for(int j=WINDOW_SIZE-2;j>=0;j--)
	{
		for(int i=0;i<max_number_detectors;i++)
		{
			int row=R0_array[i][WINDOW_SIZE-1];
			int col=R0_array[i][j];
			if((row==-1)||(col==-1))
```

```
                                {
                                        //NOTHING
                                }else
                                {

                                    c_array.set(  ((row-
1)*NUM_SYS_CALLS)+col+(NUM_SYS_CALLS*NUM_SYS_CALLS*array_num)  );
                                }
                        }
                        array_num++;
                }


                //------------------------------------------------------------------------------------------
                // TESTING  with negative detectors modified form
                //------------------------------------------------------------------------------------------

                ofstream outPrintFile1000 ("tracing_neg.txt", ios::out);
                if (!outPrintFile1000)
                {
                        cerr <<" file could not be opened" <<endl;
                        exit(1);
                }
                time(&begining_time);
                outPrintFile1000 << "Begining Time= "<<begining_time<<endl;
                char filename1000[LINE_SIZE];
                strcpy_s(filename1000,"int_login_homegrown.txt");
                ifstream inClientFile1000(filename1000,ios::in);
                if (!inClientFile1000)
                {
                        cerr <<"anomalous file could not be opened" <<endl;
                        exit(1);
                }
                int number_rows_in_testing_profile1000=0;
                int number_of_pairs_in_testing_profile1000=0;
                int value1000;
                int window_sized_array1000[WINDOW_SIZE];
                for (int i=0;i<WINDOW_SIZE;i++) window_sized_array1000[i] =-1;
                for (int abc=0;abc<WINDOW_SIZE-1;abc++)
                        {
                                if(inClientFile1000 >> value1000) window_sized_array1000[abc]=value1000;
                        }
                double percentage1000;
                int mismatches1000=0;
                int row1000, col1000;
                while (inClientFile1000 >> value1000)
                {
                        window_sized_array1000[WINDOW_SIZE-1]=value1000;
                        number_rows_in_testing_profile1000++;
                        outPrintFile1000 <<"current testing input row:  ";
                        for (int def=0;def<WINDOW_SIZE;def++) outPrintFile1000 <<
window_sized_array1000[def] << " ";
                        outPrintFile1000 <<endl;

                        for (int i=0;i<WINDOW_SIZE-1;i++)
```

```
				{
					row1000=window_sized_array1000[WINDOW_SIZE-1];
					col1000=window_sized_array1000[WINDOW_SIZE-2-i];
					if ((row1000==-1)||(col1000==-1))
					{
						//NOTHING
					}else
					{
						number_of_pairs_in_testing_profile1000++;
						if (c_array.test(  ((row1000-
1)*NUM_SYS_CALLS)+col1000+(NUM_SYS_CALLS*NUM_SYS_CALLS*i)  )==1)
						{
							mismatches1000 ++;
							outPrintFile1000 <<"mismatch pair:
<"<<row1000<<","<<col1000<<"> at plain: "<<i<<endl;
						}
					}
				}
				for (int ghi=0;ghi<WINDOW_SIZE-1;ghi++)
window_sized_array1000[ghi]=window_sized_array1000[ghi+1];
				window_sized_array1000[WINDOW_SIZE-1]=-1;
			}
			int number_neg_pairs=0;
			for (int a1=0;a1<array_num;a1++)
				for (int a2=1;a2<NUM_SYS_CALLS;a2++)
					for (int a3=1;a3<NUM_SYS_CALLS;a3++)
						if (c_array.test(  ((a2-
1)*NUM_SYS_CALLS)+a3+(NUM_SYS_CALLS*NUM_SYS_CALLS*a1)  )==1)
							number_neg_pairs++;

			int number_pos_pairs=0;
			for (int a1=0;a1<array_num;a1++)
				for (int a2=1;a2<NUM_SYS_CALLS;a2++)
					for (int a3=1;a3<NUM_SYS_CALLS;a3++)
						if (b_array.test(  ((a2-
1)*NUM_SYS_CALLS)+a3+(NUM_SYS_CALLS*NUM_SYS_CALLS*a1)  )==1)
							number_pos_pairs++;


			outPrintFile1000 << "number of pos mismatches= "<<mismatches<<endl;
			outPrintFile1000 << "number of neg mismatches= "<<mismatches1000<<endl;
			outPrintFile1000 <<" number of pos pairs= "<<number_pos_pairs<<endl;
			outPrintFile1000 <<" number of neg pairs= "<<number_neg_pairs<<endl;
			outPrintFile1000 <<" number of pos detectors= "<<number_profile_rows<<endl;
			outPrintFile1000 <<" number of neg detectors= "<<max_number_detectors<<endl;
			outPrintFile1000 <<" space cost while running of pos= "<<(b_array.size()/8)<<endl;
			outPrintFile1000 <<" space cost while running of neg= "<<(c_array.size()/8)<<endl;
			outPrintFile1000 <<" space cost while saved to disk of pos=
"<<sizeof(int)*number_pos_pairs*3<<endl;
			outPrintFile1000 <<" space cost while saved to disk of neg=
"<<sizeof(int)*number_neg_pairs*3<<endl;

			//------------------------------------------------------------------------------------------
```

APPENDIX F

LOOKAHEAD PAIRS METHOD ENHANCED WITH DANGER THEORY

SAMPLE CODE

```
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//IDS MAIN . H
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "IDSSystem.h"
void main()
{
		IDSSystem IDSsystem;
		cout <<"exiting program"<<endl;
}
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//IDSSYSTEM . H ( IDS SYSTEM HEADER FILE) // CONTROLS STARTING  IDS FUNCTIONS
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef IDSSYSTEM_H
#define IDSSYSTEM_H
#include "Th2.h"
#include "Th1.h"
#include "KillerT.h"
#include "iDC.h"
#include "DC.h"
#include "Bcell.h"
#include "ExitEngine.h"
class IDSSystem
{
public:
		IDSSystem();
		~IDSSystem();
private:
		KillerT one_KillerT;
		Th1 one_Th1;
		Th2 one_Th2;
		DC one_DC;
		ExitEngine one_ExitEngine;
		Bcell one_Bcell;
		iDC one_iDC;
};
#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//IDSSYSTEM . CPP (IDS SYSTEM SOURCE FILE) CONTROLS IDS FUNCTIONS
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "IDSSystem.h"
//--------------------------------------------------------------------------
//CONSTRUCTOR
//--------------------------------------------------------------------------

IDSSystem::IDSSystem()
   :one_Th2(one_Bcell),
            one_Th1(one_Th2,one_KillerT),
            one_Bcell(one_Th1,one_ExitEngine),
            one_DC(one_Th1),
            one_iDC(one_DC,one_ExitEngine)

{
            time_t begining_time;
            cout <<"IDS started"<<endl;
            time(&begining_time);
            one_ExitEngine.set_begining_time(begining_time);

}

//--------------------------------------------------------------------------
//DESTRUCTOR
//--------------------------------------------------------------------------

IDSSystem::~IDSSystem()
{
            cout <<"IDS shut down"<<endl;
}

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//IDC . H (IDC CELL HEADER FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef IDC_H
#define IDC_H
#include "constant_values.h"
class DC;
class ExitEngine;
class iDC{
public:
            iDC(DC &, ExitEngine &);
            void admin();
            int calculate_cytokines_iDC(int , int [CYCLE_THRESHOLD], double, double, int,
                        int [CYCLE_THRESHOLD],
                       int [CYCLE_THRESHOLD], int [CYCLE_THRESHOLD],int ,
                       int [CYCLE_THRESHOLD],int [WINDOW_SIZE]);
private:
            double C_PAMP;
            double IC;
            double C_safe;
            double C_danger;
            DC &one_DC;
            ExitEngine &one_ExitEngine;
            std::bitset<SIZE> the_array;
};
#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//IDC . CPP (IDC CELL SOURCE FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

#include "iDC.h"
#include "DC.h"
#include "ExitEngine.h"
//---------------------------------------------------------------------------
//CONSTRUCTOR
//---------------------------------------------------------------------------
iDC::iDC(DC &Ref_DC, ExitEngine &Ref_ExitEngine)
:one_DC(Ref_DC),
one_ExitEngine(Ref_ExitEngine)
{
            C_PAMP=0;
            IC=0;
            C_safe=0;
            C_danger=0;

            cout<<"constrcut iDC"<<endl;
            admin();
}
//---------------------------------------------------------------------------
//CALCULATE IDC
//---------------------------------------------------------------------------
int iDC::calculate_cytokines_iDC(int user_present1, int prev_mismatches1[CYCLE_THRESHOLD],
                        double CPU_usage1,
                        double mem_usage1,         int abnormal_signal1,
                        int prev_CPU1 [CYCLE_THRESHOLD], int prev_mem1 [CYCLE_THRESHOLD],
                        int prev_abnormal1 [CYCLE_THRESHOLD],int location1,
                         int prev_IC1[CYCLE_THRESHOLD],
                        int an_array[WINDOW_SIZE])
{
            int number_mis=0;
            int identified_CPU_attack=0;
            int identified_MEM_attack=0;
            int identified_mismatch_attack=0;
            int number_IC=0;
            prev_IC1[location1]=user_present1;
            for (int i=0;i<CYCLE_THRESHOLD;i++)
            {
                        if (prev_mismatches1[i]==1)number_mis ++;
                        if (prev_IC1[i]==1) number_IC++;

            }
            if (number_IC>0)
                         IC = (double)(number_IC/CYCLE_THRESHOLD);
            else IC =0.0;
            if (number_mis>0)
            {
                        C_PAMP = (double) (number_mis/CYCLE_THRESHOLD);
                        identified_mismatch_attack=1;
            }
            else C_PAMP =0.0;


            if (abnormal_signal1 == 1)
            {
                        prev_abnormal1[location1]=1;
            }else
            {
                        prev_abnormal1[location1]=0;
            }
            double normalized_safe_CPU;
            double normalized_danger_CPU;

            double normalized_safe_mem;
            double normalized_danger_mem;

            if (CPU_usage1 <= CPU_THRESHOLD)
            {
                         normalized_safe_CPU =CPU_usage1;
                         normalized_danger_CPU =0;
                        prev_CPU1[location1]=0;
            }else
            {
                        identified_CPU_attack=1;
                        normalized_danger_CPU = CPU_usage1;
                        normalized_safe_CPU= 0;
                        prev_CPU1[location1]=1;
            }

            if (mem_usage1 <= MEM_THRESHOLD)
            {
                         normalized_safe_mem = mem_usage1;
                         normalized_danger_mem =0;
                        prev_mem1[location1]=0;

            }else
            {
                        identified_MEM_attack=1;
                        normalized_danger_mem = mem_usage1;
                        normalized_safe_mem= 0;
                        prev_mem1[location1]=1;
            }
            int prev_CPU_count=0;
            int prev_mem_count=0;
            int prev_abnormal_count=0;
            for (int i=0;i<CYCLE_THRESHOLD;i++)
            {
                        if (prev_CPU1[i]==1) prev_CPU_count ++;
                        if (prev_mem1[i]==1) prev_mem_count ++;
                        if (prev_abnormal1[i] ==1) prev_abnormal_count ++;
            }
            double normalized_CPU = double (prev_CPU_count/CYCLE_THRESHOLD);
            double normalized_mem = double (prev_mem_count/CYCLE_THRESHOLD);
            double normalized_abnormal = double (prev_abnormal_count/CYCLE_THRESHOLD);
```

```
 C_safe = ((CPU_AFFECT_ON_S * normalized_safe_CPU)+
             (MEM_AFFECT_ON_S* normalized_safe_mem) +
             (PREV_CPU_AFFECT_ON_S*(CYCLE_THRESHOLD- normalized_CPU))+
             (PREV_MEM_AFFECT_ON_S*(CYCLE_THRESHOLD-normalized_mem))+
             (PREV_ABNORMAL_DEATH_AFFECT_ON_S*
             (CYCLE_THRESHOLD-normalized_abnormal)));

 C_danger = ((CPU_AFFECT_ON_D*normalized_danger_CPU)+
             (MEM_AFFECT_ON_D*normalized_danger_mem) +
             (ABNORMAL_DEATH_AFFECT_ON_D * abnormal_signal1)+
             (PREV_CPU_AFFECT_ON_D* normalized_CPU)+
             (PREV_MEM_AFFECT_ON_D*normalized_mem)+
             (PREV_ABNORMAL_DEATH_AFFECT_ON_D*normalized_abnormal));

int attack_type;


 if ((identified_CPU_attack==0)&&(identified_MEM_attack==0)&&
(identified_mismatch_attack==0))
 {
              attack_type = iDC_ATTACK_TYPE1;
 }else if ((identified_CPU_attack==0)&&(identified_MEM_attack==0)&&
(identified_mismatch_attack==1))
 {
              attack_type = iDC_ATTACK_TYPE2;
 }
 else if((identified_CPU_attack==0)&&(identified_MEM_attack==1)&&
(identified_mismatch_attack==0))
 {
              attack_type = iDC_ATTACK_TYPE3;
 }
 else if((identified_CPU_attack==0)&&(identified_MEM_attack==1)&&
(identified_mismatch_attack==1))
 {
              attack_type = iDC_ATTACK_TYPE4;
 }
 else if((identified_CPU_attack==1)&&(identified_MEM_attack==0)&&
(identified_mismatch_attack==0))
 {
              attack_type = iDC_ATTACK_TYPE5;
 }
 else if((identified_CPU_attack==1)&&(identified_MEM_attack==0)&&
(identified_mismatch_attack==1))
 {
              attack_type = iDC_ATTACK_TYPE6;
 }
 else if((identified_CPU_attack==1)&&(identified_MEM_attack==1)&&
(identified_mismatch_attack==0))
 {
              attack_type = iDC_ATTACK_TYPE7;
 }
 else if((identified_CPU_attack==1)&&(identified_MEM_attack==1)&&
(identified_mismatch_attack==1))
```

```
        {
                attack_type = iDC_ATTACK_TYPE8;
        }


        int temp_return_value;

        temp_return_value= one_DC.calculate_cytokines_DC(C_PAMP, C_safe, C_danger, IC,
                an_array, attack_type);
        return temp_return_value;
//-----------------------------------------------------------------------------------
//ADMIN
//-----------------------------------------------------------------------------------
void iDC::admin()
{
        int value;
        ifstream inClientFile ("lookahead_values1.txt",ios::in);
        if (!inClientFile)
        {
                cerr <<"File could not be opened2" <<endl;
                exit(1);
        }
        int row1;
        int col1;
        int array_num1;
        while ((inClientFile >> row1)&&(inClientFile >> col1)&&     (inClientFile >> array_num1))
        {
                the_array.set(  ((row1-1)*NUM_SYS_CALLS)+
                col1+(NUM_SYS_CALLS*NUM_SYS_CALLS*array_num1)  );
        }
        int user_present;//boolean either present or not
        double CPU_usage;
        double mem_usage;
        int abnormal_signal =0; //boolean either used or not.
        int prev_CPU[CYCLE_THRESHOLD];
        int prev_mem[CYCLE_THRESHOLD];
        int prev_abnormal[CYCLE_THRESHOLD];
        int prev_mismatches[CYCLE_THRESHOLD];
        int prev_IC[CYCLE_THRESHOLD];
        for (int i=0;i<CYCLE_THRESHOLD;i++)
        {
                prev_CPU[i]=0;
                prev_mem[i]=0;
                prev_abnormal[i]=0;
                prev_mismatches[i]=0;
                prev_IC[i]=0;
        }
        char filename1[LINE_SIZE];
        strcpy_s(filename1,"anomalies_testing.txt");
        ifstream inClientFile1(filename1,ios::in);
        if (!inClientFile1)
        {
                cerr <<"anomalous file could not be opened" <<endl;
                exit(1);
        }
```

```
char filename222[LINE_SIZE];
strcpy_s(filename222,"CPU_MEM_usages1.txt");
ifstream inClientFile222(filename222,ios::in);
if (!inClientFile222)
{
        cerr <<"file could not be opened6" <<endl;
        exit(1);
}
ofstream outPrintFile222 ("iDC_tracing.txt", ios::out);
if (!outPrintFile222)
{
        cerr <<"int files names file could not be opened" <<endl;
        exit(1);
}
int number_rows_in_testing_profile=0;
int number_of_pairs_in_testing_profile=0;
int window_sized_array[WINDOW_SIZE];
int threshold_sized_array[WINDOW_SIZE];
for (int i=0;i<WINDOW_SIZE;i++)
{
        window_sized_array[i] =-1;
        threshold_sized_array[i]=0;
}
for (int abc=0;abc<WINDOW_SIZE-1;abc++)
        {
                if(inClientFile1 >> value) window_sized_array[abc]=value;
        }
double percentage;
int mismatches=0;
int row, col;
int mismatch_counter =0;
int location=0;
int starting_point=0;
while (inClientFile1 >> value)
{
        int there_is_a_mismatch=0;
        window_sized_array[WINDOW_SIZE-1]=value;
        number_rows_in_testing_profile++;
        outPrintFile222 <<"current testing input row:  ";
        for (int def=0;def<WINDOW_SIZE;def++) outPrintFile222 <<
                window_sized_array[def] << " ";
        outPrintFile222 <<endl;
        for (int i=0;i<WINDOW_SIZE-1;i++)
        {
                row=window_sized_array[WINDOW_SIZE-1];
                col=window_sized_array[WINDOW_SIZE-2-i];

                if ((row == 36 ) || ( col==36)) abnormal_signal=1;

                if ((row==-1)||(col==-1))
                {
                        //NOTHING
                }else
                {
                        number_of_pairs_in_testing_profile++;
                        if (the_array.test(  ((row-1)*NUM_SYS_CALLS)+
                        col+(NUM_SYS_CALLS*NUM_SYS_CALLS*i)  )==0)
                        {
                                mismatches ++;
                                if(i==0) starting_point=((row-1)*NUM_SYS_CALLS)+
                                col+(NUM_SYS_CALLS*NUM_SYS_CALLS*i);
                                there_is_a_mismatch=1;
                                outPrintFile222 <<"mismatch pair: <"<<row<<","<<col<<
                                "> at plain: "<<i<<endl;
                        }
                }
        }
        if ((there_is_a_mismatch==1)||
                ((number_rows_in_testing_profile % MOD_VALUE)==0))
        {
                prev_mismatches[mismatch_counter]=1;
                mismatch_counter ++;
                if (mismatch_counter == CYCLE_THRESHOLD-1) mismatch_counter =0;
                (inClientFile222 >> user_present);
                (inClientFile222 >> CPU_usage);
                (inClientFile222 >> mem_usage);
                int cytokine;
                cytokine= calculate_cytokines_iDC(user_present, prev_mismatches,
                CPU_usage, mem_usage, abnormal_signal,
                prev_CPU, prev_mem, prev_abnormal, location, prev_IC,window_sized_array
                );
                location++;
                if (location == CYCLE_THRESHOLD-1) location =0;
                outPrintFile222 <<"Handeled Window: ";
                for (int i=0;i<WINDOW_SIZE;i++)
                        outPrintFile222 << setw(4)<<  window_sized_array[i];
                outPrintFile222 <<endl;
                if (there_is_a_mismatch==1)outPrintFile222 << "Is a mismatch"<<endl;
                else outPrintFile222 << "Is normal"<<endl;
                outPrintFile222 << "User present: "<< user_present<<endl;
                outPrintFile222 << "CPU usage: "<< CPU_usage<<endl;
                outPrintFile222 << "Mem usage: "<< mem_usage<<endl;
                outPrintFile222 << "Is an abnormal signal: "<< abnormal_signal<<endl;
                outPrintFile222 <<"previous CPU: ";
                for (int i=0;i<CYCLE_THRESHOLD;i++)
                        outPrintFile222 << setw(4)<<  prev_CPU[i];
                outPrintFile222 <<endl;
                outPrintFile222 <<"previous memory: ";
                for (int i=0;i<CYCLE_THRESHOLD;i++)
                        outPrintFile222 << setw(4)<<  prev_mem[i];
                outPrintFile222 <<endl;
                outPrintFile222 <<"previous abnormal: ";
                for (int i=0;i<CYCLE_THRESHOLD;i++)
                        outPrintFile222 << setw(4)<<  prev_abnormal[i];
                outPrintFile222 <<endl;
                outPrintFile222 <<"previous mismatches: ";
                for (int i=0;i<CYCLE_THRESHOLD;i++)
                        outPrintFile222 << setw(4)<<  prev_mismatches[i];
                outPrintFile222 <<endl;
                outPrintFile222 <<"previous IC: ";
```

```
					for (int i=0;i<CYCLE_THRESHOLD;i++)
						outPrintFile222 << setw(4)<<  prev_IC[i];
					outPrintFile222 <<endl;
					if (cytokine==1) outPrintFile222 << "Semi"<<endl;
					else if (cytokine==2) outPrintFile222 << "Mat"<<endl;
					outPrintFile222 <<"------------------------------------"<<endl;
				}else
				{
					prev_mismatches[mismatch_counter]=0;
					mismatch_counter ++;
					if (mismatch_counter == CYCLE_THRESHOLD-1)
						mismatch_counter =0;
				}

				for (int ghi=0;ghi<WINDOW_SIZE-1;ghi++)
				window_sized_array[ghi]=window_sized_array[ghi+1];
				window_sized_array[WINDOW_SIZE-1]=-1;
			}
			outPrintFile222 << "Number of rows in testing profile= "<<
				number_rows_in_testing_profile-1<<endl;
			outPrintFile222 << "Number of pairs in testing profile= "
				<<number_of_pairs_in_testing_profile<<endl;
			percentage = ((double) mismatches / number_of_pairs_in_testing_profile)* 100.0;
			outPrintFile222 << "Number of lookahead mismatches= "<<mismatches<<endl;
			outPrintFile222 << "Percentage of mismatches (anomaly sensitivity)= "<<percentage
				<<" %"<<endl;
			one_ExitEngine.iDC_finished();
}

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//B CELL . H
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef BCELL_H
#define BCELL_H
#include "constant_values.h"
class Th1;
class ExitEngine;
class Bcell{
public:
			Bcell(Th1 &, ExitEngine&);
			void admin();
			void receive_input_from_Th2(int, int);
private:
			std::bitset<SIZE> the_array;
			int threshold_array[SIZE];
			Th1 &one_Th1;
			ExitEngine &one_ExitEngine;
};

#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//B CELL . CPP
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "Bcell.h"
```

```
#include "Th1.h"
#include "ExitEngine.h"

//----------------------------------------------------------------------------
//CONSTRUCTOR
//----------------------------------------------------------------------------

Bcell::Bcell(Th1 &Ref_Th1, ExitEngine &Ref_ExitEngine )
:one_Th1(Ref_Th1),
one_ExitEngine(Ref_ExitEngine)
{
	for (int i=0;i<SIZE;i++) threshold_array[i]=0;
	cout <<"construct Bcell"<<endl;
	admin();
}
//----------------------------------------------------------------------------
//RECEIVE INPUT FROM TH2
//----------------------------------------------------------------------------

void Bcell::receive_input_from_Th2(int answer,int start_location)
{
	if (answer==PRIME)
	{
		for (int cc=0;cc<WINDOW_SIZE;cc++)
		{
			threshold_array[start_location+cc]++;

		}
	}else
	{
		for (int cc=0;cc<WINDOW_SIZE;cc++)
		{
			if(threshold_array[cc]<MIN_THRESHOLD_VALUE)
			{
				the_array.reset(start_location+cc);
				threshold_array[start_location+cc]=0;
			}else
			{
				threshold_array[start_location+cc]--;
			}
		}
	}
}

//----------------------------------------------------------------------------
//ADMIN
//----------------------------------------------------------------------------

void Bcell::admin()

{
	int value;
	ifstream inClientFile ("lookahead_values.txt",ios::in);
	if (!inClientFile)
	{
		cerr <<"File could not be opened2" <<endl;
		exit(1);
	}
	int row1;
	int col1;
	int array_num1;
	while (inClientFile >> row1)
	{
		(inClientFile >> col1);
		(inClientFile >> array_num1);
		the_array.set(  ((row1-1)*NUM_SYS_CALLS)+
			col1+(NUM_SYS_CALLS*NUM_SYS_CALLS*array_num1)  );
	}
	char filename1[LINE_SIZE];

	strcpy_s(filename1,"anomalies_testing.txt");
	ifstream inClientFile1(filename1,ios::in);
	if (!inClientFile1)
	{
		cerr <<"anomalous file could not be opened" <<endl;
		exit(1);
	}


	ofstream outPrintFile222 ("Bcell_tracing.txt", ios::out);
	if (!outPrintFile222)
	{
		cerr <<"int files names file could not be opened" <<endl;
		exit(1);
	}
	int number_rows_in_testing_profile=0;
	int number_of_pairs_in_testing_profile=0;

	int window_sized_array[WINDOW_SIZE];
	int threshold_sized_array[WINDOW_SIZE];
	for (int i=0;i<WINDOW_SIZE;i++)
	{
		window_sized_array[i] =-1;
		threshold_sized_array[i]=0;
	}
	for (int abc=0;abc<WINDOW_SIZE-1;abc++)
		{
			if(inClientFile1 >> value) window_sized_array[abc]=value;
		}
	double percentage;
	int mismatches=0;
	int row, col;
	int mismatch_counter =0;
	int location=0;
	int starting_point=0;

	while (inClientFile1 >> value)
	{
		int there_is_a_mismatch=0;
		window_sized_array[WINDOW_SIZE-1]=value;
```

```
          number_rows_in_testing_profile++;
          outPrintFile222 <<"current testing input row:  ";
          for (int def=0;def<WINDOW_SIZE;def++) outPrintFile222 <<
                    window_sized_array[def] << " ";
          outPrintFile222 <<endl;

          for (int i=0;i<WINDOW_SIZE-1;i++)
          {
                    row=window_sized_array[WINDOW_SIZE-1];
                    col=window_sized_array[WINDOW_SIZE-2-i];
                    if ((row==-1)||(col==-1))
                    {
                              //NOTHING
                    }else
                    {
                              number_of_pairs_in_testing_profile++;
                              if (the_array.test(  ((row-1)*NUM_SYS_CALLS)+
                              col+(NUM_SYS_CALLS*NUM_SYS_CALLS*i)  )==0)
                              {
                                        mismatches ++;
                                        if(i==0) starting_point=((row-1)*NUM_SYS_CALLS)+
                                        col+(NUM_SYS_CALLS*NUM_SYS_CALLS*i);
                                        threshold_array[((row-1)*NUM_SYS_CALLS)+
                                        col+(NUM_SYS_CALLS*NUM_SYS_CALLS*i)]++;
                                        threshold_sized_array[i]=
                                                  threshold_array[((row-1)*NUM_SYS_CALLS)+
                                                  col+(NUM_SYS_CALLS*NUM_SYS_CALLS*i)];
                                        there_is_a_mismatch=1;
                                        outPrintFile222 <<"mismatch pair: <"<<row<<","<<col
                                        <<"> at plain: "<<i<<endl;

                              }

                    }
          }
          if (there_is_a_mismatch==1)
          {
                    one_Th1.receive_input_from_B(window_sized_array,
                              threshold_sized_array,starting_point);

          }
          for (int ghi=0;ghi<WINDOW_SIZE-1;ghi++)
                    window_sized_array[ghi]=window_sized_array[ghi+1];
          window_sized_array[WINDOW_SIZE-1]=-1;
}
outPrintFile222 << "Number of rows in testing profile= "<<
                    number_rows_in_testing_profile-1<<endl;
outPrintFile222 << "Number of pairs in testing profile= "<<
          number_of_pairs_in_testing_profile<<endl;
percentage = ((double) mismatches / number_of_pairs_in_testing_profile)* 100.0;
outPrintFile222 << "Number of lookahead mismatches= "<<mismatches<<endl;
outPrintFile222 << "Percentage of mismatches (anomaly sensitivity)= "<<percentage<<
          " %"<<endl;

one_ExitEngine.Bcell_finished();
```

```
}
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//DC . H (DC CELL HEADER FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef DC_H
#define DC_H
#include "constant_values.h"
class Th1;
class DC{
public:
		DC(Th1 &);
		int calculate_cytokines_DC(double, double, double, double,
				int[WINDOW_SIZE],int);
private:
		double C_csm;
		double C_semi;
		double C_mat;
		Th1 &one_Th1;
};
#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//DC . CPP (DC CELL SOURCE FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "DC.h"
#include "Th1.h"
//-------------------------------------------------------------
DC::DC(Th1 &Ref_Th1)
:one_Th1(Ref_Th1)
{
		cout <<"construct DC"<<endl;
}
//-------------------------------------------------------------
int DC::calculate_cytokines_DC(double C_PAMP, double C_safe, double C_danger, double IC,
		int asequence[WINDOW_SIZE], int type_of_attack)
{

		 C_csm = (((W_PAMP_CSM*C_PAMP)+(W_DANGER_CSM*C_danger)+
		(W_SAFE_CSM*C_safe))/
		(W_PAMP_CSM+W_DANGER_CSM+W_SAFE_CSM))*((1+IC)/2);

		 C_semi = (((W_PAMP_SEMI*C_PAMP)+(W_DANGER_SEMI*C_danger)+
				(W_SAFE_SEMI*C_safe))/
				(W_PAMP_SEMI+W_DANGER_SEMI+W_SAFE_SEMI))*((1+IC)/2);

		 C_mat = ((W_PAMP_MAT*C_PAMP)+(W_DANGER_MAT*C_danger)+
		(W_SAFE_MAT*C_safe))*((1+IC)/2);

		 int temp_attack_type=type_of_attack;
		 int temp_sequence[WINDOW_SIZE];
		 for (int ll=0;ll<WINDOW_SIZE;ll++)
				 temp_sequence[ll]=asequence[ll];

		if (C_semi > C_mat)
		{
				int temp=SUPRESS;

				one_Th1.receive_input_from_DC(temp, temp_sequence,temp_attack_type); //semi

				return SUPRESS;
			}
			else
			{
				int temp=PRIME;

				one_Th1.receive_input_from_DC(temp, temp_sequence,temp_attack_type);//mat

				return PRIME;

			}

}
//---------------------------------------------------------------------------

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//TH1 . H (HELPER TH1 HEADER FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef TH1_H
#define TH1_H
#include "constant_values.h"
class Th2;
class KillerT;
class Th1{
public:
	Th1(Th2 &, KillerT &);
	void receive_input_from_DC(int , int [WINDOW_SIZE] , int);
	void receive_input_from_B(int [WINDOW_SIZE],int [WINDOW_SIZE],int);
	void admin();
private:
	int cyto;
	int DC_sequence[WINDOW_SIZE];
	int B_sequence[WINDOW_SIZE];
	int B_threshold_seq[WINDOW_SIZE];
	int B_seq_starting_point;
	int received_B_input;
	int DC_input;
	int attack_type;
	Th2 &one_Th2;
	KillerT &one_KillerT;
};
#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//TH1 . CPP (HELPER TH1 SOURCE FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "Th1.h"
#include "Th2.h"
#include "KillerT.h"

//---------------------------------------------------------------------------
//CONSTRUCTOR
```

```
//---------------------------------------------------------------------------

Th1::Th1(Th2 &Ref_Th2, KillerT &Ref_KillerT)
        :one_Th2(Ref_Th2),
        one_KillerT(Ref_KillerT),
        received_B_input(0),
        DC_input(0),
        cyto(0),
        B_seq_starting_point(0),
        attack_type(0)
{

        for (int hh=0;hh<WINDOW_SIZE;hh++)
        {
                DC_sequence[hh]=0;
                B_sequence[hh]=0;
                B_threshold_seq[hh]=0;
        }
}

//---------------------------------------------------------------------------
//RECEIVE INPUT FROM B CELL
//---------------------------------------------------------------------------

void Th1::receive_input_from_B(int sequence_from_B[WINDOW_SIZE],int
threshold_sequence_from_B[WINDOW_SIZE],
 int starting_point)
{
        received_B_input=1;
        for (int i=0;i<WINDOW_SIZE;i++)
        {
                B_sequence[i]=sequence_from_B[i];
                B_threshold_seq[i]=threshold_sequence_from_B[i];
        }
        B_seq_starting_point=starting_point;
        admin();
}
//---------------------------------------------------------------------------
//ADMIN
//---------------------------------------------------------------------------

void Th1::admin()
{
                int counter=0;
                int B_attacked=0;
                if ((received_B_input ==1)&&(DC_input ==0))
                {

                        for (int i=0;i <WINDOW_SIZE;i++)
                                if (B_threshold_seq[i]>MAX_THRESHOLD_VALUE) counter++;
                        if(counter>0)
                        {
                                B_attacked=1;
                                one_KillerT.receive_input(B_sequence,ATTACK_TYPE2,
                                iDC_ATTACK_TYPE1);
                        }
                        counter=0;
                        received_B_input =0;
                }else if ((received_B_input ==0)&&(DC_input ==1))
                {

                        if (cyto==PRIME)
                        {
                                one_KillerT.receive_input(DC_sequence, ATTACK_TYPE1,
                                        attack_type);
                                DC_input=0;

                        }else if ((cyto == SUPRESS)&&(B_attacked==0))// B threshold is low
                        {
                                        int same=0;
                                        for (int i=0;i<WINDOW_SIZE;i++)
                                                if (DC_sequence[i]==B_sequence[i])same ++;
                                        if (same>0)          one_Th2.receive_input(SUPRESS,
                                                B_seq_starting_point);
                        }

                } else if ((DC_input==1)&&(received_B_input==1))
                {

                        if (cyto==PRIME)
                        {
                                one_KillerT.receive_input(DC_sequence, ATTACK_TYPE3,
                                        attack_type);
                                int same=0;
                                for (int i=0;i<WINDOW_SIZE;i++)
                                        if (DC_sequence[i]==B_sequence[i])same ++;
                                if (same>0)          one_Th2.receive_input(PRIME,
                                        B_seq_starting_point);

                        }else if (cyto == SUPRESS)
                        {
                                int same=0;
                                for (int i=0;i<WINDOW_SIZE;i++)
                                        if (DC_sequence[i]==B_sequence[i])same ++;
                                if (same>0)          one_Th2.receive_input(SUPRESS,
                                        B_seq_starting_point);

                        }


                        DC_input=0;
                        received_B_input=0;
                }
}
//---------------------------------------------------------------------------
//RECEIVE INPUT FROM DC CELL
//---------------------------------------------------------------------------

void Th1::receive_input_from_DC(int value1, int value2[WINDOW_SIZE], int value3)
{
```

```
            DC_input=1;

            cyto=value1;
            for (int ii=0;ii<WINDOW_SIZE;ii++)
                        DC_sequence[ii]=value2[ii];
            attack_type= value3;
            admin();
}
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//TH2 . H   (HELPER TH2 HEADER FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef TH2_H
#define TH2_H
#include "constant_values.h"
class Bcell;
class Th2{
public:
            Th2(Bcell &);
            void receive_input(int,int);
private:
            Bcell &one_Bcell;
};
#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//TH2 . CPP (HELPER TH2 SOURCE FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "Th2.h"
#include "Bcell.h"
//---------------------------------------------------------------------------
Th2::Th2(Bcell &Ref_Bcell)
:one_Bcell(Ref_Bcell)
{
            cout <<"constrcut th2"<<endl;
}
//---------------------------------------------------------------------------
void Th2::receive_input(int cytokine_type, int B_seq_starting_point)
{
            one_Bcell.receive_input_from_Th2(cytokine_type,B_seq_starting_point);
}

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//KILLERT . H (KILLER T CELL HEADER FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef KILLERT_H
#define KILLERT_H
#include "constant_values.h"
class KillerT{
public:
            KillerT();
            void receive_input(int [WINDOW_SIZE], int , int );
};
#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//KILLERT . CPP (KILLER T CELL SOURCE FILE)
```

```
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "KillerT.h"
//---------------------------------------------------------------------------
//CONSTRUCTOR
//---------------------------------------------------------------------------

KillerT::KillerT()
{
        cout <<"constrcut killer t"<<endl;
}
//---------------------------------------------------------------------------
//RECEIVE INPUT
//---------------------------------------------------------------------------
void KillerT::receive_input(int sequence[WINDOW_SIZE], int attacktype1, int attacktype2)
{
        cout <<"Sequence: ";
        for (int mm=0;mm<WINDOW_SIZE;mm++)
                cout <<setw(4)<<sequence[mm];
        cout <<"    Generated following problems."<<endl;
        if (attacktype1 == ATTACK_TYPE2)
        {
                cout <<"Attack identified by Bcells only and no danger signals is sensed"<<endl;
        }
        else if (attacktype1==ATTACK_TYPE1)
        {
                cout <<"Attack identified by DC only and no deviaion in sys call sequences identified"<<
                                endl;
        }else if (attacktype1==ATTACK_TYPE3)
        {
                cout <<"Attack idenified by both DC and B cells"<<endl;
        }
        if ((attacktype1==ATTACK_TYPE1)||(attacktype1==ATTACK_TYPE3))
        {
                switch (attacktype2)
                {
                        case iDC_ATTACK_TYPE1:
                                cout << " DC identified following attacks CPU 0 MEM 0 Mismatch 0"
                                <<endl;
                                break;
                        case iDC_ATTACK_TYPE2:
                                cout << " DC identified following attacksCPU 0 MEM 0 Mismatch 1"
                                <<endl;
                                break;
                        case iDC_ATTACK_TYPE3:
                                cout << " DC identified following attacksCPU 0 MEM 1 Mismatch 0"
                                <<endl;
                                break;
                        case iDC_ATTACK_TYPE4:
                                cout<< " DC identified following attacksCPU 0 MEM 1 Mismatch 1"
                                <<endl;
                                break;
                        case iDC_ATTACK_TYPE5:
                                cout << " DC identified following attacks CPU 1 MEM 0 Mismatch 0"
                                <<endl;
                                break;
```

```
                    case iDC_ATTACK_TYPE6:
                        cout << " DC identified following attacks CPU 1 MEM 0 Mismatch 1"
                        <<endl;
                        break;
                    case iDC_ATTACK_TYPE7:
                        cout << " DC identified following attacks CPU 1 MEM 1 Mismatch 0"
                        <<endl;
                        break;
                    case iDC_ATTACK_TYPE8:
                        cout << " DC identified following attacks CPU 1 MEM 1 Mismatch 1"
                        <<endl;
                        break;
                }
        }

}
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//EXITENGINE . H (EXIT ENGINE HEADER FILE)
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#ifndef EXITENGINE_H
#define EXITENGINE_H
#include "constant_values.h"
class ExitEngine{
public:
        ExitEngine();
        void Bcell_finished();
        void iDC_finished();
        void set_begining_time(time_t);
private:
        int Bfinished;
        int iDCfinished;
        time_t begining_time;
        time_t ending_time;
};
#endif
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
//EXITENGINE . CPP (EXIT ENGINE SOURCE FILE) HANDLES EXITING IDS
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
#include "ExitEngine.h"

ExitEngine::ExitEngine()
:Bfinished(0),
iDCfinished(0)
{
        cout <<"construct exit engine"<<endl;
}
//------------------------------------------------------------------------
void ExitEngine::Bcell_finished()
{
        Bfinished=1;
        if (iDCfinished ==1)
        {


                time(&ending_time);
```

```
			ofstream outPrintFile ("times.txt", ios::out);
			if (!outPrintFile)
			{
				cerr <<"int files names file could not be opened" <<endl;
				exit(1);
			}
			outPrintFile << "Begining Time= "<<begining_time<<endl;
			outPrintFile << "Ending Time= "<<ending_time<<endl;
			outPrintFile << "Total testing Time= "<<ending_time-begining_time<<"
seconds"<<endl<<endl;
		}
}
//-------------------------------------------------------------------------
void ExitEngine::iDC_finished()
{
		iDCfinished=1;
		if (Bfinished==1)
		{

			time(&ending_time);
			ofstream outPrintFile ("times.txt", ios::out);
			if (!outPrintFile)
			{
				cerr <<"int files names file could not be opened" <<endl;
				exit(1);
			}
			outPrintFile << "Begining Time= "<<begining_time<<endl;
			outPrintFile << "Ending Time= "<<ending_time<<endl;
			outPrintFile << "Total testing Time= "<<ending_time-begining_time<<"
seconds"<<endl<<endl;
		}
}
//-------------------------------------------------------------------------
void ExitEngine::set_begining_time(time_t aTime)
{
		begining_time=aTime;
}
```

APPENDIX G

SAMPLE LOG FILE OF RUNNING LOOKAHEAD-PAIRS METHOD BASED IDS

In this appendix sample of the log file created while running lookahead pairs method IDS is presented. The data here show portions of the content of the log file. In particular we print the number of rows generated before removing redundant entries (rows) and then the content of these rows. Then we print the number of entries after removing redundant rows and the entries are then displayed.
Then pairs are generated and displayed as their respective locations are set to one (ON). The log file print the pair and its plane or array set. Then finally testing begins by displaying the sequence currently under investigation and highlight if a mismatch if found in this string.

```
Start of training
number_profile_rows before removing redundant rows= 363
array content with pyramid data at end of array
90 125 106 5 90 6 5 3 90 6 5 3 90 6 125 5 3
125 106 5 90 6 5 3 90 6 5 3 90 6 125 5 3 90
106 5 90 6 5 3 90 6 5 3 90 6 125 5 3 90 6
5 90 6 5 3 90 6 5 3 90 6 125 5 3 90 6 125
90 6 5 3 90 6 5 3 90 6 125 5 3 90 6 125 5
6 5 3 90 6 5 3 90 6 125 5 3 90 6 125 5 3
5 3 90 6 5 3 90 6 125 5 3 90 6 125 5 3 90
3 90 6 5 3 90 6 125 5 3 90 6 125 5 3 90 6
90 6 5 3 90 6 125 5 3 90 6 125 5 3 90 6 125
6 5 3 90 6 125 5 3 90 6 125 5 3 90 6 125 91
5 3 90 6 125 5 3 90 6 125 5 3 90 6 125 91 125
3 90 6 125 5 3 90 6 125 5 3 90 6 125 91 125 136
90 6 125 5 3 90 6 125 5 3 90 6 125 91 125 136 49
6 125 5 3 90 6 125 5 3 90 6 125 91 125 136 49 24
125 5 3 90 6 125 5 3 90 6 125 91 125 136 49 24 47
5 3 90 6 125 5 3 90 6 125 91 125 136 49 24 47 50
3 90 6 125 5 3 90 6 125 91 125 136 49 24 47 50 67
90 6 125 5 3 90 6 125 91 125 136 49 24 47 50 67 27
6 125 5 3 90 6 125 91 125 136 49 24 47 50 67 27 67
125 5 3 90 6 125 91 125 136 49 24 47 50 67 27 67 97
5 3 90 6 125 91 125 136 49 24 47 50 67 27 67 97 122
3 90 6 125 91 125 136 49 24 47 50 67 27 67 97 122 45
90 6 125 91 125 136 49 24 47 50 67 27 67 97 122 45 5
6 125 91 125 136 49 24 47 50 67 27 67 97 122 45 5 106
125 91 125 136 49 24 47 50 67 27 67 97 122 45 5 106 6
.
.
```

```
.
.6 106 67 23 12 2 67 114 67 5 108 90 3 6 91 76 75
106 67 23 12 2 67 114 67 5 108 90 3 6 91 76 75 24
67 23 12 2 67 114 67 5 108 90 3 6 91 76 75 24 102
23 12 2 67 114 67 5 108 90 3 6 91 76 75 24 102 13
12 2 67 114 67 5 108 90 3 6 91 76 75 24 102 13 20
2 67 114 67 5 108 90 3 6 91 76 75 24 102 13 20 4
67 114 67 5 108 90 3 6 91 76 75 24 102 13 20 4 6
114 67 5 108 90 3 6 91 76 75 24 102 13 20 4 6 76
67 5 108 90 3 6 91 76 75 24 102 13 20 4 6 76 75
5 108 90 3 6 91 76 75 24 102 13 20 4 6 76 75 91
108 90 3 6 91 76 75 24 102 13 20 4 6 76 75 91 1
-1 90 125 106 5 90 6 5 3 90 6 5 3 90 6 125 5
-1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90 6 125
-1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90 6
-1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90
-1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3
-1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5
-1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6
-1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90
-1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125

number_profile_rows after removing redundant rows= 357
90 125 106 5 90 6 5 3 90 6 5 3 90 6 125 5 3
125 106 5 90 6 5 3 90 6 5 3 90 6 125 5 3 90
106 5 90 6 5 3 90 6 5 3 90 6 125 5 3 90 6
5 90 6 5 3 90 6 5 3 90 6 125 5 3 90 6 125
90 6 5 3 90 6 5 3 90 6 125 5 3 90 6 125 5
6 5 3 90 6 5 3 90 6 125 5 3 90 6 125 5 3
5 3 90 6 5 3 90 6 125 5 3 90 6 125 5 3 90
3 90 6 5 3 90 6 125 5 3 90 6 125 5 3 90 6
90 6 5 3 90 6 125 5 3 90 6 125 5 3 90 6 125
6 5 3 90 6 125 5 3 90 6 125 5 3 90 6 125 91
5 3 90 6 125 5 3 90 6 125 5 3 90 6 125 91 125
3 90 6 125 5 3 90 6 125 5 3 90 6 125 91 125 136
90 6 125 5 3 90 6 125 5 3 90 6 125 91 125 136 49
6 125 5 3 90 6 125 5 3 90 6 125 91 125 136 49 24
125 5 3 90 6 125 5 3 90 6 125 91 125 136 49 24 47
5 3 90 6 125 5 3 90 6 125 91 125 136 49 24 47 50
3 90 6 125 5 3 90 6 125 91 125 136 49 24 47 50 67
90 6 125 5 3 90 6 125 91 125 136 49 24 47 50 67 27
6 125 5 3 90 6 125 91 125 136 49 24 47 50 67 27 67
125 5 3 90 6 125 91 125 136 49 24 47 50 67 27 67 97
5 3 90 6 125 91 125 136 49 24 47 50 67 27 67 97 122
.
.
.
4 6 76 75 5 67 3 67 6 106 67 23 12 2 67 114 67
6 76 75 5 67 3 67 6 106 67 23 12 2 67 114 67 5
```

```
76 75 5 67 3 67 6 106 67 23 12 2 67 114 67 5 108
75 5 67 3 67 6 106 67 23 12 2 67 114 67 5 108 90
5 67 3 67 6 106 67 23 12 2 67 114 67 5 108 90 3
67 3 67 6 106 67 23 12 2 67 114 67 5 108 90 3 6
3 67 6 106 67 23 12 2 67 114 67 5 108 90 3 6 91
67 6 106 67 23 12 2 67 114 67 5 108 90 3 6 91 76
6 106 67 23 12 2 67 114 67 5 108 90 3 6 91 76 75
106 67 23 12 2 67 114 67 5 108 90 3 6 91 76 75 24
67 23 12 2 67 114 67 5 108 90 3 6 91 76 75 24 102
23 12 2 67 114 67 5 108 90 3 6 91 76 75 24 102 13
12 2 67 114 67 5 108 90 3 6 91 76 75 24 102 13 20
2 67 114 67 5 108 90 3 6 91 76 75 24 102 13 20 4
67 114 67 5 108 90 3 6 91 76 75 24 102 13 20 4 6
114 67 5 108 90 3 6 91 76 75 24 102 13 20 4 6 76
67 5 108 90 3 6 91 76 75 24 102 13 20 4 6 76 75
5 108 90 3 6 91 76 75 24 102 13 20 4 6 76 75 91
108 90 3 6 91 76 75 24 102 13 20 4 6 76 75 91 1
-1 90 125 106 5 90 6 5 3 90 6 5 3 90 6 125 5
-1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90 6 125
-1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90 6
-1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90
-1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3
-1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5
-1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6
-1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90
-1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125
the locations that are set to 1:
<3,5> at plane: 0
<90,3> at plane: 0
<6,90> at plane: 0
<125,6> at plane: 0
<5,125> at plane: 0
<3,5> at plane: 0
<90,3> at plane: 0
<6,90> at plane: 0
<125,6> at plane: 0
<91,125> at plane: 0
<125,91> at plane: 0
<136,125> at plane: 0
<49,136> at plane: 0
<24,49> at plane: 0
<47,24> at plane: 0
<50,47> at plane: 0
<67,50> at plane: 0
<27,67> at plane: 0
<67,27> at plane: 0
<97,67> at plane: 0
<122,97> at plane: 0
<45,122> at plane: 0
```

```
<5,45> at plane: 0
<106,5> at plane: 0
<6,106> at plane: 0
<54,6> at plane: 0
<108,54> at plane: 0
<106,108> at plane: 0
<5,106> at plane: 0
<55,5> at plane: 0
<45,55> at plane: 0
<141,45> at plane: 0
.
.
.,91> at plane: 1
<49,125> at plane: 1
<24,136> at plane: 1
<47,49> at plane: 1
<50,24> at plane: 1
<67,47> at plane: 1
<27,50> at plane: 1
<67,67> at plane: 1
<97,27> at plane: 1
<122,67> at plane: 1
<45,97> at plane: 1
<5,122> at plane: 1
<106,45> at plane: 1
<6,5> at plane: 1
<54,106> at plane: 1
<108,6> at plane: 1
<106,54> at plane: 1
<5,108> at plane: 1
<55,106> at plane: 1
<45,5> at plane: 1
<141,55> at plane: 1
<106,45> at plane: 1
<6,141> at plane: 1
<57,106> at plane: 1
<54,6> at plane: 1
<16,57> at plane: 1
<15,54> at plane: 1
<54,16> at plane: 1
<67,15> at plane: 1
<111,54> at plane: 1
<67,67> at plane: 1
<66,111> at plane: 1
<5,67> at plane: 1
<6,66> at plane: 1
<63,5> at plane: 1
.
.
.
<91,90> at plane: 2
<106,6> at plane: 2
<5,125> at plane: 2
<90,91> at plane: 2
<6,106> at plane: 2
```

```
<5,5> at plane: 2
<3,90> at plane: 2
<90,6> at plane: 2
<6,5> at plane: 2
<125,3> at plane: 2
<5,90> at plane: 2
<3,6> at plane: 2
<90,125> at plane: 2
<6,5> at plane: 2
<125,3> at plane: 2
<91,90> at plane: 2
<106,6> at plane: 2
<5,125> at plane: 2
<90,91> at plane: 2
<6,106> at plane: 2
<5,5> at plane: 2
<3,90> at plane: 2
<90,6> at plane: 2
<6,5> at plane: 2
.
.
.
.
.
<23,13> at plane: 15
<12,6> at plane: 15
<2,102> at plane: 15
<67,13> at plane: 15
<114,20> at plane: 15
<67,4> at plane: 15
<5,6> at plane: 15
<108,76> at plane: 15
<90,75> at plane: 15
<3,5> at plane: 15
<6,67> at plane: 15
<91,3> at plane: 15
<76,67> at plane: 15
<75,6> at plane: 15
<24,106> at plane: 15
<102,67> at plane: 15
<13,23> at plane: 15
<20,12> at plane: 15
<4,2> at plane: 15
<6,67> at plane: 15
<76,114> at plane: 15
<75,67> at plane: 15
<91,5> at plane: 15
<1,108> at plane: 15
Begining Time= 1196614999
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3
current testing input row:  -1 -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90
current testing input row:  -1 -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6
current testing input row:  -1 -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5
current testing input row:  -1 -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3
current testing input row:  -1 -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90
current testing input row:  -1 -1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90 6
current testing input row:  -1 -1 90 125 106 5 90 6 5 3 90 6 5 3 90 6 125
current testing input row:  -1 90 125 106 5 90 6 5 3 90 6 5 3 90 6 125 5
current testing input row:  90 125 106 5 90 6 5 3 90 6 5 3 90 6 125 5 3
current testing input row:  125 106 5 90 6 5 3 90 6 5 3 90 6 125 5 3 90
current testing input row:  106 5 90 6 5 3 90 6 5 3 90 6 125 5 3 90 6
current testing input row:  5 90 6 5 3 90 6 5 3 90 6 125 5 3 90 6 125
current testing input row:  90 6 5 3 90 6 5 3 90 6 125 5 3 90 6 125 5
current testing input row:  6 5 3 90 6 5 3 90 6 125 5 3 90 6 125 5 3
current testing input row:  5 3 90 6 5 3 90 6 125 5 3 90 6 125 5 3 90
current testing input row:  3 90 6 5 3 90 6 125 5 3 90 6 125 5 3 90 6
current testing input row:  90 6 5 3 90 6 125 5 3 90 6 125 5 3 90 6 125
current testing input row:  6 5 3 90 6 125 5 3 90 6 125 5 3 90 6 125 91
current testing input row:  5 3 90 6 125 5 3 90 6 125 5 3 90 6 125 91 125
current testing input row:  3 90 6 125 5 3 90 6 125 5 3 90 6 125 91 125 136
current testing input row:  90 6 125 5 3 90 6 125 5 3 90 6 125 91 125 136 49
current testing input row:  6 125 5 3 90 6 125 5 3 90 6 125 91 125 136 49 24
current testing input row:  125 5 3 90 6 125 5 3 90 6 125 91 125 136 49 24 47
current testing input row:  5 3 90 6 125 5 3 90 6 125 91 125 136 49 24 47 50
.
.
.
.
current testing input row:  13 6 102 13 20 4 6 76 75 5 67 3 67 6 106 67 23
current testing input row:  6 102 13 20 4 6 76 75 5 67 3 67 6 106 67 23 12
current testing input row:  102 13 20 4 6 76 75 5 67 3 67 6 106 67 23 12 2
current testing input row:  13 20 4 6 76 75 5 67 3 67 6 106 67 23 12 2 67
current testing input row:  20 4 6 76 75 5 67 3 67 6 106 67 23 12 2 67 114
current testing input row:  4 6 76 75 5 67 3 67 6 106 67 23 12 2 67 114 11
mismatch pair: <11,114> at plain: 0
mismatch pair: <11,67> at plain: 1
mismatch pair: <11,2> at plain: 2
mismatch pair: <11,12> at plain: 3
mismatch pair: <11,23> at plain: 4
mismatch pair: <11,67> at plain: 5
mismatch pair: <11,106> at plain: 6
mismatch pair: <11,6> at plain: 7
mismatch pair: <11,67> at plain: 8
mismatch pair: <11,3> at plain: 9
mismatch pair: <11,67> at plain: 10
mismatch pair: <11,5> at plain: 11
mismatch pair: <11,75> at plain: 12
mismatch pair: <11,76> at plain: 13
mismatch pair: <11,6> at plain: 14
mismatch pair: <11,4> at plain: 15
current testing input row:  6 76 75 5 67 3 67 6 106 67 23 12 2 67 114 11 67
mismatch pair: <67,11> at plain: 0
mismatch pair: <67,114> at plain: 1
mismatch pair: <67,2> at plain: 3
mismatch pair: <67,12> at plain: 4
mismatch pair: <67,23> at plain: 5
current testing input row:  76 75 5 67 3 67 6 106 67 23 12 2 67 114 11 67 5
mismatch pair: <5,11> at plain: 1
mismatch pair: <5,114> at plain: 2
mismatch pair: <5,2> at plain: 4
mismatch pair: <5,12> at plain: 5
mismatch pair: <5,23> at plain: 6
mismatch pair: <5,75> at plain: 14
mismatch pair: <5,76> at plain: 15
current testing input row:  75 5 67 3 67 6 106 67 23 12 2 67 114 11 67 5 108
mismatch pair: <108,11> at plain: 2
mismatch pair: <108,114> at plain: 3
mismatch pair: <108,2> at plain: 5
mismatch pair: <108,12> at plain: 6
mismatch pair: <108,23> at plain: 7
mismatch pair: <108,75> at plain: 15
current testing input row:  5 67 3 67 6 106 67 23 12 2 67 114 11 67 5 108 90
mismatch pair: <90,11> at plain: 3
mismatch pair: <90,114> at plain: 4
mismatch pair: <90,2> at plain: 6
mismatch pair: <90,12> at plain: 7
mismatch pair: <90,23> at plain: 8
current testing input row:  67 3 67 6 106 67 23 12 2 67 114 11 67 5 108 90 3
mismatch pair: <3,11> at plain: 4
mismatch pair: <3,114> at plain: 5
mismatch pair: <3,67> at plain: 6
mismatch pair: <3,2> at plain: 7
mismatch pair: <3,12> at plain: 8
.
.
.
.
mismatch pair: <11,24> at plain: 8
mismatch pair: <11,75> at plain: 9
mismatch pair: <11,76> at plain: 10
mismatch pair: <11,91> at plain: 11
mismatch pair: <11,6> at plain: 12
mismatch pair: <11,3> at plain: 13
mismatch pair: <11,90> at plain: 14
mismatch pair: <11,108> at plain: 15
Number of system calls handeled while testing= 1350
Ending Time= 1196614999
Total testing Time= 0 seconds

Number of lookahead mismatches= 1098
Percentage of mismatches (anomaly sensitivity)= 5.11554 %
 Maximum number of lookahead-pairs SETS= 17 sets
 Minimum number of lookahead-pairs SETS= 15 sets
 Number of lookahaead pairs= 5384
Number of sets (planes)= 16planes. Each is a 256 x 256 bit array and NUM_SYS_CALLS=256
Space cost of profile while at running time= 131072 bits. = 16384 bytes.
Space cost of profile while saved to disk= 21536 bytes
```

APPENDIX H

SAMPLE LOG FILE OF THE OUPUT PRODUCED WHEN TESTING CASE 10 WITH THE LOOKAHEAD-PAIRS METHOD ENHANCED WITH DANGER THEORY.

The following is a sample log file of the output produced when testing the lookahead-pairs method enhanced with danger theory IDS. The ouput is produced when testing the system on case 10 where CPU and memory usages are normal but a number of contigous mismataches occur. The number of allowable mismatches is determined by the system adminstrator in advance. In our system the mismatch threshold is equal to 10. The system will continue to produce a "semi" or normal behavior output, as shown in the following 9 sections of the output, and then start producing "mat" or intrusive behavior output afterwards.

```
current testing input row:  90 125 106 5
mismatch pair: <5,106> at plain: 0
mismatch pair: <5,125> at plain: 1
mismatch pair: <5,90> at plain: 2
Handeled Window:   90 125 106   5
Is a mismatch
User present: 1
CPU usage: 20.5
Mem usage: 30.3
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   0   0   0   0   0   0   0   0   0
previous IC:    1   0   0   0   0   0   0   0   0   0
Semi
------------------------------------
current testing input row:  125 106 5 90
mismatch pair: <90,5> at plain: 0
mismatch pair: <90,106> at plain: 1
mismatch pair: <90,125> at plain: 2
Handeled Window:  125 106   5  90
Is a mismatch
User present: 1
CPU usage: 21.3
Mem usage: 25.2
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   0   0   0   0   0   0   0   0
previous IC:    1   1   0   0   0   0   0   0   0   0
Semi
------------------------------------
current testing input row:  106 5 90 6
mismatch pair: <6,90> at plain: 0
mismatch pair: <6,5> at plain: 1
mismatch pair: <6,106> at plain: 2
Handeled Window:  106   5  90   6
Is a mismatch
User present: 1
CPU usage: 20.3
Mem usage: 30.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   0   0   0   0   0   0   0
previous IC:    1   1   1   0   0   0   0   0   0   0
Semi
------------------------------------
current testing input row:  5 90 6 5
mismatch pair: <5,6> at plain: 0
mismatch pair: <5,90> at plain: 1
mismatch pair: <5,5> at plain: 2
Handeled Window:    5  90   6   5
Is a mismatch
User present: 1
CPU usage: 21.3
Mem usage: 31.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   0   0   0   0   0   0
previous IC:    1   1   1   1   0   0   0   0   0   0
Semi
------------------------------------
current testing input row:  90 6 5 3
mismatch pair: <3,5> at plain: 0
mismatch pair: <3,6> at plain: 1
mismatch pair: <3,90> at plain: 2
Handeled Window:   90   6   5   3
Is a mismatch
User present: 1
CPU usage: 22.3
Mem usage: 30.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   0   0   0   0   0
previous IC:    1   1   1   1   1   0   0   0   0   0
```

```
Semi
------------------------------------
current testing input row:  6 5 3 90
mismatch pair: <90,3> at plain: 0
mismatch pair: <90,5> at plain: 1
mismatch pair: <90,6> at plain: 2
Handeled Window:    6   5   3  90
Is a mismatch
User present: 1
CPU usage: 21.7
Mem usage: 33.2
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   0   0   0   0
previous IC:    1   1   1   1   1   1   0   0   0   0
Semi
------------------------------------
current testing input row:  5 3 90 20
mismatch pair: <20,90> at plain: 0
mismatch pair: <20,3> at plain: 1
mismatch pair: <20,5> at plain: 2
Handeled Window:    5   3  90  20
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 30.2
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   0   0   0
previous IC:    1   1   1   1   1   1   1   0   0   0
Semi
------------------------------------
current testing input row:  3 90 20 6
mismatch pair: <6,20> at plain: 0
mismatch pair: <6,90> at plain: 1
Handeled Window:    3  90  20   6
Is a mismatch
User present: 1
CPU usage: 20.1
Mem usage: 31.4
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   0   0
previous IC:    1   1   1   1   1   1   1   1   0   0
Semi
------------------------------------
current testing input row:  90 20 6 3
mismatch pair: <3,6> at plain: 0
mismatch pair: <3,20> at plain: 1
mismatch pair: <3,90> at plain: 2
Handeled Window:   90  20   6   3
Is a mismatch
User present: 1
CPU usage: 20.5
Mem usage: 30.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   0
previous IC:    1   1   1   1   1   1   1   1   1   0
Semi
------------------------------------
current testing input row:  20 6 3 89
mismatch pair: <89,3> at plain: 0
mismatch pair: <89,6> at plain: 1
mismatch pair: <89,20> at plain: 2
Handeled Window:   20   6   3  89
Is a mismatch
User present: 1
CPU usage: 19.6
Mem usage: 28.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  6 3 89 33
mismatch pair: <33,89> at plain: 0
mismatch pair: <33,3> at plain: 1
mismatch pair: <33,6> at plain: 2
Handeled Window:    6   3  89  33
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  3 89 33 19
mismatch pair: <19,33> at plain: 0
mismatch pair: <19,89> at plain: 1
mismatch pair: <19,3> at plain: 2
Handeled Window:    3  89  33  19
Is a mismatch
User present: 1
```

```
CPU usage: 20.1
Mem usage: 29.5
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  89 33 19 20
mismatch pair: <20,33> at plain: 1
mismatch pair: <20,89> at plain: 2
Handeled Window:   89  33  19  20
Is a mismatch
User present: 1
CPU usage: 20.9
Mem usage: 29.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  33 19 20 33
mismatch pair: <33,20> at plain: 0
mismatch pair: <33,19> at plain: 1
mismatch pair: <33,33> at plain: 2
Handeled Window:   33  19  20  33
Is a mismatch
User present: 1
CPU usage: 21.4
Mem usage: 28.2
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  19 20 33 1
mismatch pair: <1,33> at plain: 0
mismatch pair: <1,20> at plain: 1
mismatch pair: <1,19> at plain: 2
Handeled Window:   19  20  33   1
Is a mismatch
User present: 1
CPU usage: 22.1
Mem usage: 30.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  20 33 1 4
mismatch pair: <4,1> at plain: 0
mismatch pair: <4,33> at plain: 1
mismatch pair: <4,20> at plain: 2
Handeled Window:   20  33   1   4
Is a mismatch
User present: 1
CPU usage: 22.3
Mem usage: 35.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  33 1 4 29
mismatch pair: <29,4> at plain: 0
mismatch pair: <29,1> at plain: 1
mismatch pair: <29,33> at plain: 2
Handeled Window:   33   1   4  29
Is a mismatch
User present: 1
CPU usage: 21.9
Mem usage: 34.8
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  1 4 29 3
mismatch pair: <3,29> at plain: 0
mismatch pair: <3,4> at plain: 1
mismatch pair: <3,1> at plain: 2
Handeled Window:    1   4  29   3
Is a mismatch
User present: 1
CPU usage: 21.6
Mem usage: 34.8
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
```

```
------------------------------------
current testing input row:  4 29 3 7
mismatch pair: <7,3> at plain: 0
mismatch pair: <7,29> at plain: 1
Handeled Window:    4  29   3   7
Is a mismatch
User present: 1
CPU usage: 20.1
Mem usage: 34.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  29 3 7 38
mismatch pair: <38,7> at plain: 0
mismatch pair: <38,3> at plain: 1
mismatch pair: <38,29> at plain: 2
Handeled Window:   29   3   7  38
Is a mismatch
User present: 1
CPU usage: 19.9
Mem usage: 33.1
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  3 7 38 20
mismatch pair: <20,38> at plain: 0
mismatch pair: <20,7> at plain: 1
mismatch pair: <20,3> at plain: 2
Handeled Window:    3   7  38  20
Is a mismatch
User present: 1
CPU usage: 19.9
Mem usage: 30.2
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  7 38 20 90
mismatch pair: <90,20> at plain: 0
mismatch pair: <90,38> at plain: 1
mismatch pair: <90,7> at plain: 2
```

```
Handeled Window:    7  38  20  90
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  38 20 90 106
mismatch pair: <106,90> at plain: 0
mismatch pair: <106,20> at plain: 1
mismatch pair: <106,38> at plain: 2
Handeled Window:   38  20  90 106
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  20 90 106 29
mismatch pair: <29,106> at plain: 0
mismatch pair: <29,90> at plain: 1
mismatch pair: <29,20> at plain: 2
Handeled Window:   20  90 106  29
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  90 106 29 33
mismatch pair: <33,29> at plain: 0
mismatch pair: <33,106> at plain: 1
mismatch pair: <33,90> at plain: 2
Handeled Window:   90 106  29  33
Is a mismatch
User present: 1
CPU usage: 21.1
```

```
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  106 29 33 22
mismatch pair: <22,33> at plain: 0
mismatch pair: <22,29> at plain: 1
mismatch pair: <22,106> at plain: 2
Handeled Window:  106  29  33  22
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  29 33 22 40
mismatch pair: <40,22> at plain: 0
mismatch pair: <40,33> at plain: 1
mismatch pair: <40,29> at plain: 2
Handeled Window:   29  33  22  40
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  33 22 40 19
mismatch pair: <19,40> at plain: 0
mismatch pair: <19,22> at plain: 1
mismatch pair: <19,33> at plain: 2
Handeled Window:   33  22  40  19
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  22 40 19 29
mismatch pair: <29,19> at plain: 0
mismatch pair: <29,40> at plain: 1
mismatch pair: <29,22> at plain: 2
Handeled Window:   22  40  19  29
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  40 19 29 3
mismatch pair: <3,29> at plain: 0
mismatch pair: <3,19> at plain: 1
mismatch pair: <3,40> at plain: 2
Handeled Window:   40  19  29   3
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
current testing input row:  19 29 3 44
mismatch pair: <44,3> at plain: 0
mismatch pair: <44,29> at plain: 1
mismatch pair: <44,19> at plain: 2
Handeled Window:   19  29   3  44
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
```

```
------------------------------------
current testing input row:  29 3 44 1
mismatch pair: <1,44> at plain: 0
mismatch pair: <1,3> at plain: 1
mismatch pair: <1,29> at plain: 2
Handeled Window:   29   3  44   1
Is a mismatch
User present: 1
CPU usage: 21.1
Mem usage: 29.9
Is an abnormal signal: 0
previous CPU:    0   0   0   0   0   0   0   0   0   0
previous memory:    0   0   0   0   0   0   0   0   0   0
previous abnormal:    0   0   0   0   0   0   0   0   0   0
previous mismatches:    1   1   1   1   1   1   1   1   1   1
previous IC:    1   1   1   1   1   1   1   1   1   0
Mat
------------------------------------
Number of rows in testing profile= 31
Number of pairs in testing profile= 96
Number of lookahead mismatches= 93
Percentage of mismatches (anomaly sensitivity)= 96.875 %
```

APPENDIX I

PATTERNS GENERATED BY THE VARIABLE-LENGTH WITH OVERLAP RELATIONSHIP BASED IDS

Number of patterns in training database: 32
Max pattern length: 43
Min pattern length: 2
Average pattern length: 22
Space cost of profile while at running time= 206848 bytes
Space cost of profile while saved to disk= 1360 bytes
The following table shows the different patterns generated by our system.

3 19
3 6 13 20 4 6 76 75 5 67 3 67 6 106 67 23 12 2 67 114 67
3 6 13 20 4 6 76 75 102 13 4 5 67 3 67 6 106 108 90 54 4 67 23 12 2 67 114
5 108
5 45 108 90 3
5 81
6 125 91 3
6 91
13 5
13 54 13 4 54 3 54 4
13 6 54 108
13 4 6

16 15 46 49
27 24 50 71 70 33 23 70 71 20 5 3 13 19 4 6 5 67 27 143 27 4 143 6 5 19 3 5 3 6 13 108 90 54 4 19 13 4 6
76 75 24 76 75
76 75 24 54 108
76 75 24 13 20 4 6 76 75 91 1
76 75 24 102 13 20 4 6 76 75 91 1
90 3 19 6 91 13 5 13 76 75 5 108
90 6 5 3
90 6 125 5 3
90 3 106 5
90 6 125 91 106 5
90 3 6 91 76 75 24 5 108
90 125 106 5
90 6 125 91 125 136 49 24 47 50 67 27 67 97 122 45 5 106 6 54 108 106 5 55 45 141 106 6 57 54 16 15 54 67 111 67 66 5 6 63 6 54 106
90 19
106 5 55 141
106 6 5 3
106 6 5 3 13 6 102 13 20 4 6 76 75 5 67 3 67 6 106 67 23 12 2 67 114 67
106 6 5 3 13 6 102 13 20 4 6 76 75 102 13 4 5 67 3 67 6 106 4 67 23 12 2 67 114
106 6 5 3 13 6 102 13 20 4 6 76 75 102 13 4 5 67 3 67 6 106 4 67 23 12 2 67 114 67