MODELING THE ARTIFICIAL IMMUNE SYSTEM

TO THE HUMAN IMMUNE SYSTEM

WITH THE USE OF AGENTS

By
BASHAR BARRISHI
Bachelor of Science
Computer Science
Mutah University
Karak, Jordan
1994

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2004

MODELING THE ARTIFICIAL IMMUNE SYSTEM

TO THE HUMAN IMMUNE SYSTEM

WITH THE USE OF AGENTS

Thesis Approved:

Blayne Mayfield

Thesis Advisor

George Hedrick

Venkatesh Sarangan

A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my great appreciation to my thesis advisor, Dr. Mayfield for his insight, intelligence, constructive guidance and support. My sincere appreciation extends to my committee members Dr. Hedrick and Dr. Sarangan for their guidance, assistance and encouragement. I wish to express my appreciation to Dr. Samadzadeh for his support, guidance and valuable counseling. I wish to thank my friend and colleague Mr. Cain for his support and encouragement.

I would like to express my sincere appreciation to my father and mother for their support and blessings, you were always my inspiration. Thanks to my sister and brother for their support and love.

My special gratitude and appreciation is to my leader, role model, friend and King, Abdullah II Bin Al-Hussein. It was your support, trust, and inspiration that helped me through my achievements. My beloved King, thank you.

Finally, I would like to thank the department of computer science and the OSU library faculty, staff, and students for their support. Thanks to all my family and friends. To all those that I forgot to thank, I hope you will find it in your heart to forgive.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

ABSTRACT

In a continuous search for computer security, researchers and software developers are trying to provide all the possible means to stop the threat to information. These efforts include providing different ways to save the data, backup techniques, and security applications against harmful programs.

This research discusses the use of the human (biological) immune system to provide the basis of a model for an artificial immune system. The research lays out a complete workflow for the model with the use of agents and commercial off-the-shelf products to work together to counter malware. The novel idea in this work is in providing a self-healing system that is outside the software architectural descriptive handling (while avoiding the weaknesses of the current models); which utilizes the achievements of the market security products to provide a complete self-healing package from malware in a controlled environment.

This research provides the ground for a complete implementation of a product which can handle programs considered untrustworthy. The agent uses techniques such as sandboxing, processing priority and bandwidth control to quarantine the malicious code and prevent it from spreading.

I. INTRODUCTION

1.1 The Problem.

Malicious software (*malware*) has been a threat to organizations and personal machines since the first written computer program. In fact, any program not debugged correctly may cause a problem of some kind or undesirable result. This code, if executed in such a context, will have an unacceptable output. A common definition among the security community of malware is: “Short for malicious software, which is designed specifically to damage or to disrupt a system, such as a virus or a Trojan horse” [15]. This definition is debated all over the security research community, even when it is used by the community members; where it lacks complete definition. If someone writes code that overwrites a part of the bootable hard drive in a silent way (i.e. without informing the user of the running process), that code is not malware for the writer, because he[1] knows the complete functionality of the code, he might be using it to fix a problem. If someone other than the code writer gets a copy of that code and executes it without knowing all of its functionality, then according to the definition above, this code could be considered malware because it damaged and disrupted the system.

The goal of this study is to present an architecture for a security system that provides a self-healing capability with minimal user interference. The significance is that such an architecture will provide the system the capability to identify malware based on its characteristics and behavior. This prevention results in saving time and money.

[1] Whenever “He” is used in the text, it means “He/She.”

A complete and precise definition of malware is difficult to find; due to the changing nature of such code. This research focuses on the hidden functionality of the malware, defining it as a piece of code (single program or an application) that performs an undesirable action without the user's knowledge or without the user's knowledge of all its actions.

This definition helps to describe the code functionality in this approach, which includes viruses, worms and the like. Our concern also includes malware that activates without the user's knowledge or acceptance, such as *spyware*. The result of running such applications may be destructive or a mere violation of privacy.

The easy access and wide usage of the Internet make it a prime target for malicious activity. In particular, the Internet has become a powerful mechanism for propagating malicious software programs designed to annoy (e.g., deface web pages), spread misinformation (e.g., false news reports or stock quotes), deny service (e.g., corrupt hard disks), steal financial information (e.g. credit card numbers), and enable remote login (e.g., Trojan horses). The two most popular ways to spread such malicious software are commonly referred to as *worms* (such as Code Red) and *email viruses* (such as Melissa or Love Bug). However, it is increasingly difficult to categorize malicious software programs using these terms [11].

Mapping an artificial model to a biological model is a helpful method used in the study of evolutionary computation. Approaches such as *Artificial Life* study biological life to achieve optimization, just as ant colonies are studied to find different approaches to problems like the Traveling Salesman problem. With this in mind, researchers began looking for a biological model to help in achieving better security. Researchers used the

Human Immune System (HIS) as the biological model, which is the same model in all the vertebrates, due to its complexity and its familiar functionality. The HIS model can help us better understand the biological approach for handling an antigen attack, and in trying to develop a solution in the artificial system similar to the biological process to handle security threats.

Building components that approximate the behavior of the human immune system components to encounter malware without depending on the user interference may provide an enhanced self-healing capability. This paper proposes a self-healing architecture based on the HIS model with the use of *agents* to protect the system from malware threats. The agents coordinate the actions and behaviors of the security components, perform anomaly detection, identify unknown malware and make/suggest decisions by communicating with the administrator, here, the administrator intervention is welcomed, but not necessary.

1.2 Purpose of the Study.

The purpose of the study is to provide a working model of a self-healing system that helps in preventing from malware threats, inspired by the human immune system working architecture, which proved to be effective in handling antigens unknown to the system. The components of the model are mostly commercial off-the-shelf, except for the agent that manages these components and coordinates their behavior once the agent detects an anomaly in the system.

In researching a security self-healing system that models the HIS, accuracy in the behavior model is required to design future adaptation or modification if needed, based on the original model. For an example, the HIS does not behave in an intelligent manner during the detection or the cleaning phases. Therefore, the Artificial Immune System (AIS) follows a step-by-step sequence in detecting and cleaning antigens, based on trial and error, depending on the information that it gets from its previous experiences, which is logged in a database for future use. Meanwhile, the components may behave intelligently based on their duties and functionalities, depending on the provider’s design of the product.

The AIS will behave in a natural manner by quarantining the infected node from the rest of the network to avoid spread of the “disease”, by utilizing strict firewall policies. Alongside the prevention of antigen spread, it is required to protect the system from the external threats and perpetrators awaiting the necessary conditions to perform harmful actions.

II. LITERATURE REVIEW

2.1 Brief Description of the Human Immune System.

The human immune system is extremely complex. Some believe it has evolved over hundreds of millions of years to respond to invasion of the pathogenic microbes that regularly attempt to infect our bodies and the invasion of the microbes that tried to infect our genetic ancestors. There are similarities between the immune system of humans and those of the most primitive of vertebrates [6], with the difference in the complexity and nature of the threats they may encounter in their environments.

Diseases in the body are recognized by *symptoms* that occur as the result of a weakened immune system not being able to produce sufficient antibodies to counter a particular strain of virus, bacteria or fungus. The symptoms could be pain, swelling, infected wounds, poor digestion, stiff joints, weakened bones or general debilitation.

The immune system does not rely on one single mechanism to deter invaders, but instead uses many strategies, the most important of which are detailed below. The main division between the strategies is that between *innate immunity*, which does not require previous exposure to the invading microbe, and *acquired immunity*, whereby the immune system "remembers" how to deal with a microbe that it has dealt with before [1].

Phagocytes (white blood cells) are the soldiers of the immune system, and provide innate immunity. They are responsible for swallowing, killing and digesting invading microbes. The process of swallowing microbes is known as phagocytosis. There are two main types of phagocyte:

- Microphages. These cells are also known as Polymorphonuclear Leucocytes, PMNs and Polymorphs. These cells start life in the bone marrow. They are constantly circulating in the blood. They cannot replicate, and live for only a few days. The bone marrow contains large reserves of microphages.
- Macrophages. These cells start out life as monocytes, which originate in the stem cells in the bone marrow, but when they are first called into action, they turn into macrophages. Macrophages are not as numerous as microphages, and there are no large reserves of them, but they are longer lived than microphages. Macrophages are stationed at strategic locations throughout the body, usually in places that are not otherwise well defended. These areas include the alveoli of the lungs, the abdominal (peritoneal) and chest (pleural) cavities, under the top layer of the skin and the intestines. Macrophages are the front line of defense against microbial invasion in these areas.

Macrophages engulf antigens, process them internally, and then display parts of them on their surface together with some of their own proteins. This sensitizes the T cells to recognize these antigens. All cells are coated with various substances. Clusters of Differentiation (CD) encompass more than one hundred and sixty clusters, each of which is a different chemical molecule that coats the surface.

An immunocompetent but as yet immature B-lymphocyte is stimulated to maturity when an antigen binds to its surface receptors and there is a T helper cell nearby (to release a cytokine). This sensitizes or primes the B cell and it undergoes clonal selection (Appendix A1), which means it reproduces asexually by mitosis. Most of the family of clones becomes plasma cells. These cells, after an initial lag, produce highly specific antibodies at a rate of as many as 2000 molecules per second for four to five days. The other B cells become long-lived memory cells. (Appendix ‘A’ describes the interaction between the T-cells, B-cells and the antigens [6]).

When these tasks are complete, the Macrophages have one further task to complete. They return to the lymph nodes, displaying the remnants of the destroyed invader on their surface. This has the effect of stimulating the cells of the acquired immunity system into action.

Immunity can be either natural or artificial, innate or acquired/adaptive, and either active or passive.

- Active natural (contact with infection): develops slowly, is long term, and antigen-specific.
- Active artificial (immunization): develops slowly, lasts for several years, and is specific to the antigen for which the immunization was given.
- Passive natural (transplacental, i.e. mother to child): develops immediately, is temporary, and affects all antigens to which the mother has immunity.
- Passive artificial (injection of gamma globulin): develops immediately, is temporary, and affects all antigens to which the donor has immunity.

The goal of all *vaccines* is to promote a primary immune reaction so that when the organism is again exposed to the antigen, a much stronger secondary immune response will be elicited. Any subsequent immune response to an antigen is called a secondary response and it has the following properties:

- Shorter lag time,
- More rapid buildup,

- Higher overall level of response,
- More specific or better "fit" to the invading antigen.

In summary, the following innate immunity will be the model for the digitized immune system. It is comprised of a collection of proteins that "recognize" corresponding proteins on the cell walls of invading microbes. When such invading microbes have been recognized, the following actions are taken:

- The "alarm" is sounded. Chemicals, known as chemotaxins, that attract phagocytes are emitted. This process is known as chemotaxis. The phagocytes follow the trail of chemotaxins to arrive at the site of invasion.
- The invading bacteria are "marked" with chemicals that make them stand out. These chemicals are known as opsonins, from the Latin word *opsonium*, meaning "sauce". This "marking" greatly increases the chances of the invading bacteria being phagocytosed.

2.2 Related Work on Artificial Immune Systems.

Zhang et al. [16] discuss an immunity-based model for *network intrusion detection*, which could be looked at as the simplest way of applying the Human Immune System (HIS) model on a computer world. The simplicity lies in the fact that these immunity agents will have steady and fixed targets on which to focus, which are the incoming ports and the packets being transferred. The paper focuses on certain aspects of the HIS such as the threshold of the number of detected antigens to provide activation for killer cells, also the memory model to attack the already known antigens. The paper describes the characteristics of the Intrusion Detection (ID) system as *distributed*, *self-organized* and *lightweight*. It also discusses and identifies the nature of self versus alien connection in correlation to *trusted* and *not-trusted* transactions. The task is performed through two processes, the *detector generation,* and the *detection process*. The detectors are generated randomly and considered immature at this stage. If the detector produces false positives during the training period, it will be eliminated. The detectors use the pattern matching to provide detection of non-self transactions, which are generated outside the network/organization entity. However, the process in which the generation of the detectors is being used was not well documented because it describes the propagation of the characteristics to newly generated detectors without defining the mechanism for achieving such a goal. In the paper it seems that there is a trivial way of generating a detector, training and testing it albeit sometimes immaturely. However, if a detector fails it will be eliminated, with very little use of its good characteristics.

In Robert et al. [12], the paper discusses a distributed architecture for virus detection, which should ease the processing pressure (detection and job execution) of the individual nodes. This is achieved by protecting the node from infection and assigning the tasks allocated to the infected node to other nodes for execution. It also introduces the concept of self (files generated within the system) or non-self (files from an external source) identification methods presented in [5] to identify individual files posing threats, or infected files. Some self-files might mutate to non-self by being infected. Another approach described in the paper is the use of virus *decoy programs* to attract viruses to infect them. Once the file is being infected, the virus signature is extracted and monitored.

The authors also describe a *leveled approach* to the problem: local, network and global. Each level has its responsibilities and functionalities. In the local level decoys used to attract the attackers are deployed (entrapment) and identification is performed on the antigen. Every attack or malware characteristic is added to the knowledge base for future reference. The network area performs the classification and reporting, while the global level performs the strategy management and selection of decoys based on an evolutionary algorithm approach, in which the fitness function is based on the attracting characteristics of the decoy and its success.

Regina et al. [9] present an excellent approximation between the HIS and the Artificial Immune System (AIS). The study discusses innate and acquired immunity, the *stages of a disease*, and the countermeasures that are taken to eliminate the threat. In their correlation the authors discuss four main aspects that both systems have to monitor: availability, correction, integrity and accountability. Table (1) displays the correlation and their duties.

	Human Immune System	Artificial Immune System
Integrity	This is a way to guarantee that the genetic codes present in the cells will not be corrupted by any pathogen	Data has to be protected from intentional or accidental corruption
Availability	This aspect allows the body to continue working even under attack of a pathogen	Information, such as the computer, must be accessible when necessary and as desired
Correction	This mechanism prevents the immune system against attacks of the (body) cells	False alarms from an incorrect classification of computational events must be minimized
Accountability	These are means adopted by the immune system to identify, find and destroy the pathological agents	The security system must be configured to preserve sufficient information from the intrusion that can be permitted to trace the origin of the attack
Confidentiality	There are no concepts of secret data or confidential information	Data access must only be allowed to authorized users

Table 1. Correlation between the HIS and the AIS [9]

The main threat the paper discusses is the intrusion into the system. The tools described to detect the threat are: (1) *misuse intrusion*, where well-defined attacks are launched against known system vulnerability, and (2) *anomaly intrusions*, which can be identified as activities based on deviation from normal system usage patterns. The paper classifies four possible scenarios that the system might encounter, checked against four *log files*: Hacking, violations, violations-ignore, and ignore. The hacking log file would be composed of keywords that characterize a hacking attempt. If detected, the immune system is activated. The violations log file is composed of keywords that characterize a violation. If detected, protective actions are required, i.e., blocking certain ports. The violations-ignore log file contains keywords opposite of the violations log check. Activity is logged and no action is required. The ignore log file holds keywords that are safe for the user. The paper displays experimental results of improved numbers of true positive detection and lesser numbers of false positives, compared to previously used signature approaches.

Neil et al. [10] describe *JISYS* system, which is AIS for network security. The main activities of the system are to collect information about the data transferred and categorize it as self (initialization). Performing this frequently "generalizes" on these data sets to identify non-self. Over time it *mutates* to apprehend new changes (accepted ones), to generate new patterns and generations, then to generalize on the new data again. The process will repeat for all the data sets. This process continues to produce AIS to recognize any data sets that are not legitimate to traffic the network.

The drawback of the system as described by the authors is the extensive execution time that the system takes [The complexity of the algorithm is $O(n^2)$ in the worst case], due to the need for processing the data transferred and then matching it with the library and mutating if required. The system was developed to identify *fraud in audited data*, but it is generalized to monitor network traffic as well.

Taheri et al. [14] introduce two approaches to utilize *agent federation* for error control by imitating the HIS. Agent federation is a process of generating numerous numbers of agents with different characteristics abstracted from a knowledgebase. The approaches discussed are the immune recruitment mechanism (IRM) and the genetic algorithms (GA). With IRM, the antibodies are selected from a pool randomly. These agents work to locate and handle. The agents are called to counter the antigen once it is located. The federation of agents chooses the *individual* with the best capabilities to achieve the required task. If it succeeds then it will be *cloned* and will spread to destroy the remaining antigen. If no individual from the initial set possesses the required capability then all the initial set will be terminated and a new population will be generated. The process of selection is repeated until a successful clone is found. Once a clone is selected and the cleansing is completed then the characteristics of that clone are *saved in a memory* location for future reference and its capabilities will be within the potentially cloned agents (phase space theory).

The GA approach is based on *fitness*, *mutation* and *regeneration*. Where an initial population is generated, the fitness function (based on the type of threat) selects the best fit members of the population. This is a basic operation in genetic algorithm, which selects the best members of a group based on certain criteria. The desirable characters in these members are forwarded to the next generation and so on until a certain number of iterations are performed or the desired set of characters is met (desired set of genes in the chromosome). The process is repeated as it checks for the fitness until the near optimum is reached or a mutation is applied to certain members. In a search for the near optimum individual, that will be used to achieve the required cleansing of the antigen by reproducing the fittest members. After a successful encounter, the system activates a process of recording the gene “chromosome” in memory to be generated in the future in case a similar threat occurs.

Other researches (i.e. [38], [39], [40]) discuss the approximation between the HIS and the AIS by tackling certain aspects of both systems, but none has approached the use of available security tools (either COTS or customized) or the use of agents to orchestrate their work to achieve the required immunity.

2.3 Models of malware.

The main threat of malware comes from the intentions of its creator; some code might be created to be a joke while others target core information to delete or encrypt. Regardless of the intentions or hidden goals, the best thing for the administrator is to monitor all applications, as it is his right of knowing what is happening on his machine, and his privileges of privacy.

According to the institute of certified security associates (ICSA) labs 7th annual prevalence survey in 2001, conducted on a group of 300 organizations had 1,182,634 encounters on 666,327 machines during the 20 months of the survey period from January 2000 through August 2001. This translates to 113 encounters per 1,000 machines per month over the entire survey period. Global infection rates calculated from the surveys of 1996 through 2001 showed a significant annual growth rate of approximately 20 encounters per month per 1,000 PCs for each year in that period. In 2000, 36% of those reporting disasters estimated that servers were down one hour or less. By contrast, 65% of the year 2001 respondents reported downtime of one hour or less, with 53% claiming no server downtime at all. More than 80% of those reporting a disaster required 20 persons-days or less to recover from their virus disasters. The median response was four person-days for recovery. On average, this cost between $5,500 (median) and $69,000 (average) in estimated direct costs [17].

Based on its target, malware can be classified in many categories. Here is an explanation of some of its types and characteristics [4]:

1. *Virus*: Code that attaches itself to other software, such as a patch algorithm, by redirecting the original starting point of the host to the start point of the virus, or by residing on the machine and pointing certain hosts to point-execute the virus when activated. Replicates and attempts to attach itself to other applications. Effect varies from humorous to catastrophic. Might attack boot sector in Microsoft operating systems, *Terminate and Stay Resident* (TSR) in the memory (until certain conditions hold). Also, might attack applications, or network protocols. Meanwhile, in Unix, the threats of malware are a little different, and will be discussed later in this section. Certain viruses mutate or use stealth techniques to escape Antivirus (AV) software. They may attack any platform, though most of them are platform dependent. Prevention can be achieved by limiting connectivity to other machines. Movement of host means a spread of the virus, where the host could be a file or a mobile machine such as a laptop with wireless capability. Common symptoms could be physically monitored, such as files expanding, date/time stamps changing, slow computer due to the virus using the resources to multiply, or system failure. Common

countermeasures include identification/ containment/ recover (if possible) of the host, using AV software.

2. *Worm*: A stand alone application that prefers network environments, often designed to propagate through networks, generally targeting multitasking machines with open network standard. Infected systems perform slower than usual and might fail. Monitored connectivity is a very good tool for prevention. Identification, containment and system recover is the countermeasure with use of AV software.
3. *Trojan Horse*: Might be a virus or a worm. Main target is to gain the user's cooperation by mocking a useful program to get access to certain information. Then, the Trojan relays the targeted information back to perpetrator. The best prevention is knowledge and training for users. Trojans are executed by the user, and may not reproduce, but they may target a specific machine or group of users. They may stay idle for a long time waiting for execution of the program, so users must be alerted not to execute it and take evasive actions if possible.
4. *Time/Logic Bomb*: A virus or a worm that activates according to certain time-stamp/conditions. They mostly try to cause damage by spreading and multiplying as much as possible. These take time to deploy, so careful monitoring will allow detection before the trigger. Team effort and user awareness are important prevention and detection tools, and user's concerns should be noticed. Contain, identify and recover are countermeasures with use of AV to help assure cleansing.
5. *Rabbits*: These target multi-tasking systems, draining the system resources by multiplying until complete system failure occurs. They act like worms on all machine levels consuming CPU, network, disk and memory resources. Firewalls play a major

role in prevention besides anomaly identifiers. Initial symptoms are the same as viruses, with increasing slowness to total paralysis.

6. *Back Doors and OS vulnerabilities*: Some programmers leave an open door in the applications that they write to allow debugging space and direct access points. These might be exploited by hackers or perpetrators to gain access or to perform other actions they may desire. Nothing is perfect, including the operating systems (OS). Every now and then patches are sent to fix certain problems and vulnerabilities in the OS. These fixes are not always forced on the user's machine. Therefore, some users might take more time before applying it to their machines. The actual declaration of the fix is an announcement of a problem, and certain perpetrators might use such a bug to attack the users who have not yet applied the fix. Users should be notified of any new updates and of the importance of applying these fixes. When applications are released, backdoors should be sealed. If there is a severe need to leave the back door in the released version, it should be protected and the administrator should be alarmed when it is used.

7. *Spyware*: These are applications that are installed on the machine with the user's approval (in most cases). However the spyware might do more than what was advertised, such as telling the weather forecast while reporting surfing behavior or information about the system. The important violation here is the privacy of the user and his right to know the behavior of the application, which it should not do such actions without the user's approval. Certain applications might have been installed legitimately with the user's consent, but the applications have the hidden

functionality of serving a third party, such as reporting other software activity on the machine (i.e. monitoring licensing).

8. *UNIX malware*: UNIX shell scripting malware is considered one of the major weaknesses in the UNIX/Linux environment; it may control program configuration and start/kill services. Bourne (sh), Bourne Again (Bash), Korn, C and Tops C shell scripting could be used as interpreters, or a completely new shell script could be created with a simple tempting name like "runme.sh"[18]. Creating malware using shell scripting is relatively easy. Simple viruses may be very short, consisting of only a few lines, and even less code is needed to construct a Trojan. Another threat in the UNIX/Linux environment is the mobile code "Javascript" that uses the Java virtual machine, which is used in both Windows and UNIX operating systems. This means that mobile malware could be executed on both machines, albeit having the virtual machine enabled. Windows emulators are also a possible drive for malware designed to operate on Windows, hence the availability of the emulator will provide the proper grounds for these malware to execute. To be fair with UNIX, it does not have the volume of attack that MS Windows has due to its levels of security, but it is still not considered immune.
9. *Macintosh malware*: Because Macintosh OS X is shipped with all its vulnerable services turned off and because Apple has a small market share, little opportunity is provided to spreading threats and malware in the Macintosh community. Experts say that this doesn't mean Macintoshes are safe, but that it is very difficult to activate a virus. No application is allowed to launch any script without the user's approval. Another plus for Macintosh is the UNIX based system that can handle multiple users

on the same machine, preventing infection from spreading from one account/user to another on the same machine. In the Macintosh environment, the most frequent way to infect a machine is to convince the user to willingly activate a script or set his machine to automatically activate scripts in email messages, or by running certain applications in Windows emulator. Infection alternatives are definitely available, but "black hats" (hackers and the like), did not have the interest in the target, yet!

In conclusion, a malware writer can persuade a victim that the code is doing something desirable, and the user will launch it. The writer can also bypass the victim by writing a self-propelled code that executes without the direct action from the user, depending on the settings of the operating system's security features.

These malware can pose a threat in one way or another to the user. Yet we have not discussed *evolving programs* that might attack in many shapes where they learn to change strategy and attack in many shapes to achieve a certain goal. Currently, this is not a wide spread threat and available technologies do not provide enough flexibility and speed to achieve such goals. These applications have neural network engines that provide alternating patterns of attack, creating either a virus or a worm, but evolving as a complete application, mutating in different shapes and techniques. [10].

2.4 Approximation Between the Human Immune System and the Available Security Systems.

When the first worm was released in 1988, available antivirus software at that time could not see it because it was a standalone application which multiplied and spread while the antivirus expected an *embedded code*. Therefore it slipped through and caused loss in tens of millions of dollars. The threat of the unknown next step currently stands.

The recurrently spreading technique for malware nowadays uses email attachments containing Worms or Trojan Horses. Mutating applications might be the new threat that combines various malware techniques, which might not be detected by the current counter measure applications. This situation dictates the daily continuing race between malware producers and Counter Measures (CM) developers. The CM applications are required to be competent to provide the required security and confidence to deliver peace of mind to the administrators. These applications require certification and updates to keep them effective against known threats. Following are some common known commercial off-the-shelf (COTS) tools for detecting and handling malware:

- *Intrusion Detection* (ID): Detects inappropriate, incorrect, or anomalous activity [16]. ID systems that operate on a host to detect malicious activity on that host are called host-based ID systems, and ID systems that operate on network data flows are called network-based ID systems. The basic approaches in ID are using statistical anomaly or pattern matching. External threats are mostly monitored and identified using ID, but certain applications may produce huge numbers of false positives.
- *Firewalls*: A firewall is a system or group of systems that enforces an access control policy between two networks. The actual means by which this is accomplished varies widely, but in principle the firewall can be thought of as a pair of mechanisms: one which exists to block traffic, and the other which exists to permit traffic. Some firewalls place a greater emphasis on blocking traffic, while others emphasize permitting traffic. Probably the most important thing to recognize about a firewall is that it implements an access control policy defined by the administrator. Simply put, policies are everything the firewall can follow.

- *Anti-Viruses* (AV): Programs used to detect and remove computer viruses by identifying their signature (i.e. binary pattern). The simplest kind of AV scans executable files and boot blocks for a list of known viruses. Others are constantly active, attempting to detect the actions of general classes of viruses. Antivirus software should always include a regular update service allowing it to keep up with the latest viruses as they are released.

- *Behavior Monitors/Blockers* [2]: Behavior blockers watch ActiveX, Java applets, various scripting languages, and other mobile code that arrives on a host via e-mail, the Internet, or other network connections. Some blockers isolate this code in a "*sandbox,*" restricting the code's access to various OS resources and applications. Other blockers insert themselves into the kernel of a host's OS to intercept system calls.

As mentioned before, the Human Immune System (HIS) has its tools to counter the threats arising from various sources. We can describe the Artificial Immune System (AIS) in the same manner, where both systems seek the following four properties: detection, diversity, learning and tolerance:

- o Detection – Detection (or recognition) of chemical components between pathogen fragments and receptors on the surface of the lymphocyte occurs in a human immune system.
- In the same manner, the AIS seeks the detection of any virus or harmful code that enters the system illegally, intends to cause damage, existing on the system without acquiring the user's approval or acceptance.

- Diversity – Detection in the immune system is related to non-self elements of the organism. Thus, the immune system must have diverse receptors to ensure that at least some lymphocytes will react to the pathogenic element. The solution adopted by the body relies on a dynamic protection via a continuous renewal of lymphocytes.

- The AIS has the same concept, with diverse tasks and frontiers, such as when ID handles the communication ports and the agent sandbox mobile code. The diversity of the threats that the AIS can handle provides protection from different sources at the same time. Hence, each element manages a different aspect of the system, handling a huge number of threats.

- Learning – The immune system must be capable of detecting as quickly as possible the pathogen and eliminating it. It includes a principle which allows lymphocytes to recognize and adapt themselves to specific foreign protein structures, and to "remember" these structures as soon as possible when needed. These principles are implemented by the B cells.

- The only task that the AIS lacks is the self (adaptive) learning, where most of the COTS products require continuous updates with patches from the producers to update them with current and new threats. The papers reviewed in the literature research discuss many ways to achieve the adaptability to new threats. This research discusses ways to achieve this goal with the use of agents.

- Tolerance – The molecules that mark a cell as a self gene are contained in the chromosome sections also known as Major Histocompatibility Complex (MHC).

- The AIS has to be tolerant with self "programs" to avoid destroying them while it has to be decisive with any alien code. It is important to have some kind of rollback or backup sequence to correct any errors that may occur during that process. It also needs to be able to fix self "programs" if infected.

As we can see, both immune and computer security systems share common security concerns. They both intend to protect their corresponding systems against attacks and intrusions that cause anomalies in the system.

2.5 Related Work on Self-Healing Systems.

Self-healing systems technology is a gray zone between self-adaptability, software engineering, software architecture and object oriented programming. In a system, the mechanical parts can be configured to perform self-healing by providing an array of redundant resources (e.g. array of hard disks), and if one of the operable HDD fails, another is loaded by the backup copy and operated instead.

To achieve the self-healing aspect in a system, there should be an automated mechanism to allow the system to get back online or back to full capacity without the involvement of the operator. Before the use of object-oriented programming, the system architecture was only available to the administrators to terminate or initiate a process manually; a failure of a function (i.e. having a bug in the code) resulted in the failure of the whole system, unless the administrator interfered. Furthermore, the systems were not built in module architecture, where processes could be altered in a single command but not applications, which required a set of operations to achieve replacement, termination or initiation. If the system failed, it required to be taken offline and rebooted to reload a fresh (good) copy of the system to return it to operation. This process is very time consuming, considering slow machines, and the massive number of peripherals attached to the system. However, a system composed of multiple components means that once a component fails, then that component can be replaced with a corrected one. These components are not necessarily sitting idle, waiting to be swapped.

As a matter of fact, the components that are the most heavily used are the ones that are quicker to develop bugs. To be able to switch these heavily used components, the hot-swapping concept is introduced. It provides an approach to load the fresh copy of the component in the workspace, and then delete all the pointers from the old copy and point them to the new one. If successful, then the old copy is removed, and the new copy is adopted; the system is back to normal.

To achieve this, the application should have the self-awareness to realize that there is a problem and adapt to the changing requirements. Software engineering has to consider the design and architecture of the components to make it easy to swap the component without halting a huge number of the system components.

Neil et al. [19] describe the essentials for the ability to dynamically repair a system based on its architecture at runtime. These are:

1. Descriptive knowledge of the current architecture;
2. Ability to express the change of the architecture;
3. Analysis of the repair expectation and its validity; and
4. Ability to execute the repair in runtime.

Software architecture description language (ADL) can describe the changes in the architecture of the system and provide an analysis of the suggested changes that could be applied as a fix. Component sensors, such as bandwidth or cycle time, can be used to describe the health of the system. Reconfiguration strategy should be considered to provide the alternative solutions.

In software frameworks, approaches for software self-awareness and self-healing are a growing demand in languages structure and design. Open object request broker (ORB) provides reflection capability of middleware which, in part, provides more knowledge of the system structure and behavior [20]. This is being applied with the use of common object request broker architecture (CORBA) interceptors and dynamic proxies in Java. Providing a component framework can help in managing the system with a better understanding of the overall state. Implementation of the open ORB provides monitors for events and quality of service, which directs the controllers that govern the strategy selectors and implementers to a better solution.

2.6 Comparative Study of Self-Healing Systems.

Systems are increasingly required to work continuously. The idea of taking a system off-line to perform either hardware or software maintenance is becoming impractical financially and professionally. Online hardware component switching and swapping is a solution for hardware malfunction. Mapping this hardware concept to software components provides a parallel work paradigm for software malfunction runtime maintenance.

User's requirements and system resources change frequently. To provide the desired quality of service and continuity of that service, a cost effective technique is required to reliably and dynamically adapt a system's behavior to provide optimal service.

The work on software adaptability is not a new topic. Runtime assertion checking and exceptions handling are simple examples of software adaptability, but these handling techniques are not capable of determining the source of the problem, and are not able to decide proper strategy handling for future errors.

Software engineering has proposed many alternatives and techniques to handle changing software environments. Research and study has been done in that field to present the performance model as the monitoring element and strategy evaluation tool. The basic concept in these studies focuses on the architecture models as the cornerstone of model-based adaptation. Oreizy et al. use a hierarchical publish-subscribe service via C2, where all communications among components occur via connectors, thus minimizing component interdependencies and strictly separating computation from communication. The style also imposes topological constraints; every component has a “top” and a “bottom” side, with a single communication port on each side. This restriction greatly simplifies the task of adding, removing, or reconnecting a component. A C2 connector also has a top and a bottom, but the number of communication ports is determined by the components attached to it. A connector can accommodate any number of components or other connectors. This enables C2 connectors to accommodate runtime rebinding. Finally, all communications among components are done asynchronously by exchanging messages through connectors [21].

An older study by Gorlick et al. uses data-flow style via Weaves. Weaves are networks of concurrently executing tool fragments that communicate by passing objects. Weaves are distinguished from other data flow styles by their emphasis on instrumentation, continuous observability, and dynamic rearrangement. Basic low-overhead instrumentation is inserted automatically. Weaves can be executed at any time by means of sophisticated analysis agents, without degrading the performance of the weave. Weaves can be dynamically snipped and spliced without interrupting the data flow; this permits novel forms of experimentation and analysis. Weaves execute efficiently on a broad spectrum of architectures and offer numerous opportunities for parallel execution [22].

Magee et al. use bi-directional communication links via Darwin, an architectural description language specifically designed for the specification and construction of distributed software systems. It deliberately separates the description of structure from that of computation and interaction in order to provide a clear separation of concerns. A Darwin architecture can be used to compose component implementations to build a system and/or to compose LTS (labeled transition systems) specifications of component behavior for system property analysis [23].

Wermelinger and Fiadeiro use architecture primitives to effect architectural changes, independent of particular styles, mainly by using graph transformation approach for software reconfiguration. Their work provides a formally based language that integrates the aspects of architectural description, constraints, and modification [24].

Using architecture modeling to achieve adaptability is the preferred approach to address this issue. In the previous papers, the architecture model is used in different ways. Those research studies are relatively old, and have been used by other researchers to build upon. The first of these approaches that this paper discusses is the graph transformation approach; it is a relatively new concept that is being approached, and there is no further research that extends it yet. The other work discussed afterwards is built on architectural models, but each uses a different technique in monitoring and applying the changes.

2.6.1 A Graph based (Re)configuration language.

Wermelinger and Fiadeiro [26] introduced in the Software Engineering ESEC/FSE’99 an algebraic software architecture reconfiguration approach [25], both heterogeneous and uniform. It is heterogeneous because it provides a dedicated, separate sub-language for each aspect: a program design language for computation, a declarative language for constraints, and an operational language for reconfiguration.

It is uniform because it uses Category Theory as a semantic foundation both for configurations taken as categorical diagrams and reconfiguration achieved through algebraic graph rewriting techniques.

The approach also provides a strict separation between computation and (re)configuration, while keeping them explicitly related; the components do not have access nor can they change the configuration variables or call scripts. The reconfiguration scripts have access but cannot change the state of components. Notice that replacing a component by another one of the same design with a different state is not what is meant by state change, because there are actually two components involved. The original one is removed and a new one is created.

The main goal of the language is to provide high-level constructs that are suited to the architectural level of description of a system. In particular, interactions are created and removed at the level of connectors, hence guaranteeing that configurations are always instances of the architecture [26].

In the literature, the authors discuss four elements that should be monitored and handled within a healing system design. These are: modification time and source, modification operation, modification constraints, and system state. To handle all these issues, a complete knowledge of the system state is required and a tool is needed to express this knowledge. A graph design of the system nodes and connections could provide expressional means of the correlations between the components. The authors use CommUnity programs to provide the syntactic layout of the architecture. CommUnity is independent of the actual data types used with pre-defined sorts and functions given by a fixed algebraic signature in the usual sense.

A CommUnity design consists of a set of type defined variables and a set of actions. There are input, output and private variables. Input variables are read-only. Output and private variables are called local variables and cannot be changed by the environment. A design with input variables is open in the sense that it needs to be connected to other components of the system to read data.

Connecting a design D_1 with a design D_2 is done through a channel: a set of bindings $i_{1,j}$ -$i_{2,j}$, where each $i_{1,j}$ is a non-private variable or a set of shared actions of design D_1. In the first case, $i_{2,j}$ must be a non-private variable of D_2, of the same sort as $i_{1,j}$, and the pair $i_{1,1}$-$i_{2,1}$ denotes that the two variables are to be shared. In the second case, $i_{2,j}$ must be a set of shared actions of D_2. Moreover, every shared action of the involved designs can appear at most once in the channel definition. Intuitively, a pair:

$\{a_{1,1}, . . . , a_{1,n}\}$-$\{a_{2,1}, . . . , a_{2,m}\}$

states that any action $a_{1,i}$ of D_1 must occur simultaneously (i.e., synchronize) with some action $a_{2,j}$ of D_2 and vice versa.

A configuration is given by an undirected labeled graph, where each node is labeled by a design and each arc by a channel, such that no node is connected to itself, and no two output variables are directly or indirectly shared.

A run-time configuration is a configuration in which each node, besides being labeled with a design *D*, is also labeled with its current state, i.e., with one pair *<l,value>* for each local variable *l* of *D*. Because local variables cannot be shared among designs, it is not possible for two different nodes to have different values for the same variable. Hence, the *colimit* (a universal categorical construction) of a run-time configuration always exists and is given by the colimit of the underlying configuration together with the disjoint union of all variable-value pairs.

A key factor for architectural description is a notion of refinement that can be used to support abstraction. A design *R* refines design *P* if each variable of *P* is mapped to a variable of *R* of the same sort and kind (input, output or private), and each action of *P* is mapped to a set of actions of *R* of the same kind (private or shared), such that the functionality and interface (i.e., the “binding points”) of *P* are preserved. The interface is preserved by requiring the mapping of input and output variables to be injective and the image of a shared action to be a non-empty set.

An architecture defines the designs that may be used as components/roles and the refinement relationships between them; the designs that may be used only for roles, the connectors, and the refinement morphisms for each role; the configuration variables; and a constraint on the possible configurations.

A common reconfiguration is to update a component (e.g., to add new functionality, eliminate bugs, or improve efficiency). This is achieved through the replacement of a component by a refinement other than the identity of it. The syntax:

[Node$_2$:=]

create Design$_2$ *as* [*Refinement*(]Node$_1$[)]

with l_1 := Exp$_1$ k . . .

where Node$_2$ is of type Design$_2$ and Node$_1$'s type is some Design. This command removes Node$_1$ and replaces it by a new node Node$_2$. For any glue to which Node$_1$ was connected, through some channel *c*, the new node becomes connected to the same glue through a channel that is the composition of *c* with Refinement.

The command to replace a component is the most complex one, because one cannot know a priori to which connectors the component to be replaced is attached. Therefore the command is compiled into a set of productions that do the replacement in three phases: The first introduces the new component, the second re-links all connectors to the new component, and the last phase removes the original component. For the second phase, there is one production for each role of each connector that the node to be replaced may initiate.

For example, if there were two connectors C$_1$(R$_1$,R$_2$) and C$_2$(R$_4$,R$_5$,R$_6$), and a node n$_1$ of type N$_1$ were to be replaced by a node n$_2$ of type N$_2$, with refinements from R$_1$, R$_4$ and R$_6$ to N$_1$, then there would be three productions; One replacing the first channel of C$_1$ to n$_1$ by a channel from C$_2$ to n$_2$, another production for the first channel of C$_2$ and the last production for the third channel of $_{C2}$. The set of productions generated for this second phase of the replacement is to be applied until no left-hand side can be matched to the current configuration. At that point, the node to be replaced has no connectors attached and the single rule for the last phase can be applied: It simply removes the node, updating the node reference given in the command.

Notes:

This approach introduces a language that provides a high level construct of the system architecture. The language is capable of configuring and reconfiguring a system architecture offline and at run-time. The interactions in the language are created and removed at the level of connectors; this guarantees that the configurations are always instances of the architecture. This design adds more constraints on the architecture. The components of the language are not reusable and they are attached to the system components. Removing or modifying any component of the language will affect the system and its components.

Another note on this approach is the transition phase in the replacement of the system component or reconfiguration. In most cases, the component might be involved in transactions or computations that are reading or writing data. The design did not address this issue because pointers and counters might be involved and because the reconfiguration might be lost.

Rolling back to the previous configuration is not discussed. Guarantees of a better reconfiguration cannot be provided, based on the hand written scripts that govern the chosen new configuration. The choice of a bad configuration could lead the language into a series of reconfigurations; it might be better to roll-back to the previous configuration and chose another scenario. This issue was not discussed in the current papers, yet the authors have the concept of evolving architecture suggested in future researches, which could be a solution to such a problem.

2.6.2 Model-Based adaptation for self-healing systems.

One of the main drives to this research paper is the need to decentralize the error checkers in the systems such as Java exceptions or RPC timeouts. They suffer from the problem that localized error handling may not be able to determine the true source of the problem, and hence the required remedial action. Moreover, while they can trap errors, they are not well-suited to recognizing “softer” system anomalies, such as gradual degradation of performance, or patterns of unreliability. They make it difficult to change adaptation policies, because they are so intertwined with the normal code of the system.

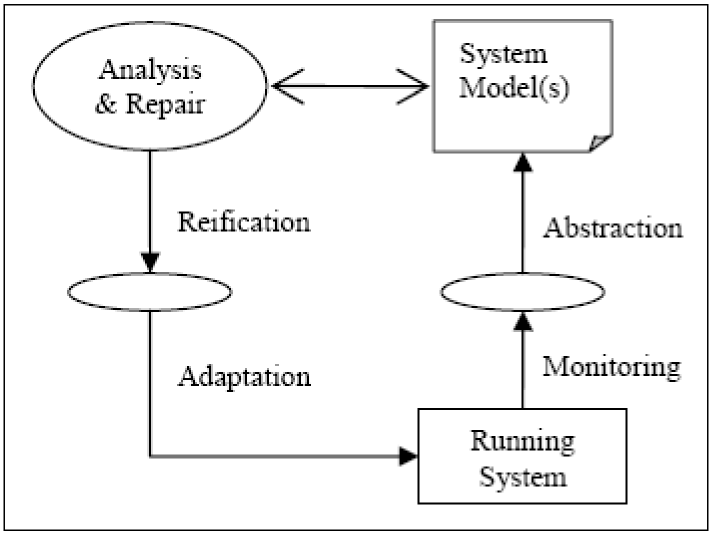


Figure 1. Externalized Adaptation In A Model Based Self-Healing System [27]

In a very rough description of the system, externalized adaptation supports a kind of closed-loop control system paradigm (Figure 1). In this paradigm, system behavior is monitored by components outside the running system. These components are responsible for determining when a system’s behavior is within the envelope of acceptable system parameters, and when it falls outside of those limits, adapting the system. To accomplish these tasks, the externalized mechanisms maintain one or more system models, which provide an abstract, global view of the running system, and support reasoning about system problems and repairs.

This approach [27] advertises the reuse of components, due to the fact that they are not localized. It also provides different models to choose from, which gives more control over the performance vs. reliability relationship. This brings the option of implementing new models easier and on demand. Security is preserved better with the priori knowledge of the system architectures available.

The centerpiece of the approach is the use of architectural models, by using a simple scheme in which an architectural model is represented as a graph of interacting components. This is the core architectural representation scheme adopted by a number of architecture description languages (ADLs), including Acme, xADL, and SADL.

To account for various behavioral properties of a system, elements in the graph can be annotated with property lists. For example, properties associated with a connector might define its protocol of interaction, or performance attributes, such as delay or bandwidth. Representing an architectural model as an arbitrary graph of generic components and connectors has the advantage of being extremely general and open ended.

An architectural style typically defines a set of types for components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may be composed. The challenge is to engineer things so that the system adapts appropriately at run time. To get information out of the running system, the model uses low-level monitoring mechanisms that refine various aspects of the executing system. This is done with the use of existing off-the-shelf performance-oriented “system probes”. (In the implementation, Remos Monitoring System is used).

To translate architectural repairs into actual system changes, the authors use a table-driven translator that interprets architectural repair operators in terms of the lower level system modifications. In the running system the monitoring mechanisms update architectural properties, causing re-evaluation of constraints. Violated constraints, such as high client-server latencies or low server loads trigger repairs, which are carried out on the architectural representation and translated into corresponding actions on the system itself. Matching the architectural style to the existing system infrastructure helps guarantee that relevant information can be extracted. The architectural changes can be propagated to the running system; architectural constraints are checked in the running system via a tool called Armani.

Notes:

Providing the tools in an externalized manner results in less overhead for the system components. It also prevents any changes to the monitoring tools from effecting the system components. Yet, the authors did not discuss the overhead on the communicating channels (monitors to the executing system, and among the adaptation system components). This overhead includes monitor probing and data analysis.

Another note on the use of Armani ADL is that it requires the administrator to handcraft the scripts for any suggested reconfigurations. Moreover it does not tolerate any mistakes or unhandled issues in the configuration. This forces the designer to write explicit constraints to prevent dangling roles, or in distributed systems, require reconfiguration scripts to connect the components port by port, for an example [30].

The proposed approach describes a monitoring and self-adapting mechanism for a system. The adaptation is not intended to reconfigure the system or change the architecture, but to tune up the components of the system. In conclusion, it is an adaptation to a changing environment. Yet the approach is not used in the actual implementation, and it does not affect the tuning up of the system. A better use of the model is to use the ADL to swap components that are causing a bottleneck due to a bug in the execution. The ADL is a good tool to reconfigure the system where a fresh copy is swapped and the component is re-located in a less active role in the new architecture. By doing so, any problems of traffic or malfunctioning could be solved.

2.6.3 The DMonA Architecture.

In this paper [28], the authors are proposing the DiPS (Distrinet Protocol Stack) component architecture as a solution to develop manageable system software such as file systems or protocol stacks. DiPS separate functionality from other aspects such as inter-component communication and internal parallelism. The paper argues that such separation of concerns makes managing specific aspects, such as concurrency, easier. The system is constructed of reusable components that adapt themselves to cope with changing circumstances.

The approach uses DMonA, the DiPS monitor and management extension architecture (Figure 2), which allows detection of performance bottlenecks, proposes solutions and deploys them at run-time. The authors focus on the run-time adaptation initiated by the system itself, using self-monitoring tools.

The main goal for DiPS is to support reuse and adaptation at design-time as well as at run-time. The DiPS component (packet) is a building block surrounded with an explicit entry (forwarder) and exit (receiver) points. Data is encapsulated within the packet. DiPs system is created by connecting packets into a pipe. A dispatcher unit dispatches requests into different branches of the pipe based on meta-information in the request.

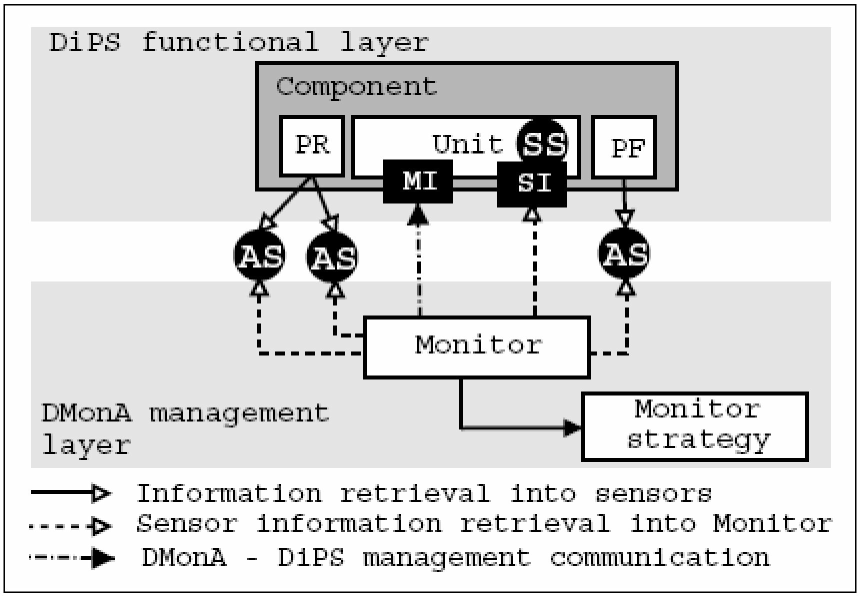

DiPS functional layer
Component
PR
Unit
SS
PF
MI
SI
AS
AS
AS
Monitor
DMonA management layer
Monitor strategy
Information retrieval into sensors
Sensor information retrieval into Monitor
DMonA - DiPS management communication

Figure 2. Communication Channels Between Dmona And Dips Layers.

DMonA is a monitoring and management extension of the DiPS component architecture. It takes advantage of existing abstractions of the packets (entry and exit points). Management can be divided in three sub tasks: detecting problems (self-monitoring), proposing solutions to the problem (self-adapting), and deploying the proposed solution (self-healing).

The primary goals of DMonA are to provide with DiPS a self-adapting system that can be applied in different contexts, with sensors and monitors that can be changed according to the context. DMonA and DiPS are orthogonal, meaning that the removal or modification of DMonA will not affect the functionality of DiPS, allowing the reusability of the DMonA components outside the DiPS system.

The sensors provide the required information from the functional level to the monitors, which are used to evaluate and manage the system. The sensors are of two types: state sensors which provide information about the internal state of the system and analysis sensors which provide information about the messages flow in the system. Based on the gathered information from the monitors, DMonA chooses the proper action to improve the system's performance.

After a problem is detected, a solution is proposed by the adaptability strategy, and then is applied. The performance is then monitored again to check the efficiency of the chosen strategy.

Notes:

The adaptability strategy that chooses the solutions for the detected problems is not well described. The paper does not explain the mechanisms of choosing the appropriate strategy or how this strategy was put together in the first place. A good approach to construct a solution is the use of xADL to provide architecture reconfiguration tools that can be monitored and adjusted based on the changing demands (environment). Keep in mind that ADL tools are not self evolving and require a hands-on updating of the changing settings, which includes script writing and management.

Another aspect the authors did not discuss is the feedback from the system on the quality of the proposed solution. A rollback technique, though may not be needed, but still in place to be included in the design, to prevent looping in search for an optimal performance. To solve those two issues, a special monitor could be added to provide a feedback to the strategy management on the initial system performance, or to provide a statistical comparison between the previous architecture and the suggested reconfiguration.

2.6.4 Kinesthetics eXtreme (KX).

This approach [29] introduces a dynamic adaptation facility. The infrastructure consists of multiple layers, with the objectives of:

1. Probing, measuring and reporting of activity and state during the execution of the target system among its components and connectors;

2. Gauging, analysis and interpretation of the reported events;

3. Whenever necessary, providing feedback onto the probes and gauges to focus them, or onto the running target system to direct its automatic adjustment and reconfiguration.

This approach to adaptation adds a feedback control loop outside and orthogonal to the legacy system's main computation, control and communication.

The first interaction the KX has with the system is through the probes that provide feedback to the monitors. The probes should have a minimally invasive approach that can be guaranteed to have zero or negligible effect on the performance and reliability of the system. A probe here is an individual sensor attached to or associated with a running program. The probe can sense some portion of the program's execution and make this data available by issuing events. Probes are reusable components that can be customized and adjusted based on the running environment.

A Smart Event XML scheme is introduced as a standard format for structuring the probes output; each probe has a unique identifier and tag structure appropriate for reuse outside the current architecture.

Gauges are software entities that gather, filter, aggregate, compute, and/or analyze measurement information about the software systems, interpreting the information gathered by the probes. Gauges operate within a framework consisting of two major components: event packager transforms raw data format from the probes output into Smart Events, and event distiller which recognizes complex temporal event patterns from multiple probe sources, and constructs higher-level measurements to reflect the system state represented by the events. The Event Distiller is "programmed" by a collection of condition/action rules.

Gauge outputs are inputs to a decision process that determines what course of action to take, if any. The decision process may be supported by a variety of tools, including an architecture transformation tool that reacts to gauges that detect differences between the running and the nominal architecture. Executing high-level repair actions such as to reconfiguring the architecture will often involve several activities at the implementation level. Some of these activities may be conditional or dependent on others, or may simply fail, so the adaptation process is expressed as a workflow rich enough to express contingency plans. This decision and control layer might also invoke the management actions of the probe and gauge layers on occasion, to induce refined measurements before proceeding with adaptation.

Workflakes is used to instantiate and coordinate all kinds of adaptations of the running system, as a decentralized workflow system. The workflow is currently expressed as a set of coding patterns in Java, which are then dynamically loaded into and executed by the Workflakes engine. The language needs to specify both sequential and parallel execution of actions, and how to deal with unsuccessful actions, by retrying, attempting alternate actions, rolling back or compensating changes.

Effector (implementation) actions cover a spectrum from simple adaptations such as relatively low-level adjustments, to a well-defined target system API (changing a process variable or calling a method), to potentially complex reconfiguration commands that cause structurally significant changes, possibly involving high-level adjustments at the system/environmental level. The latter may involve starting, migrating, restarting, or stopping one or more processes, and/or rearranging the connections among components. Workflakes currently conducts an adaptation workflow by selecting, instantiating and dispatching Worklets mobile agents (hand crafted), and coordinating the activities of the deployed Worklets on the target system's components and connectors.

Notes:

The approach mentioned in this paper has considered all the aspects of monitoring and analyzing the gathered data. It also provided a strategy, or adaptation, mechanism to choose the best reconfiguration and provide feedback on that change to adjust the gauges performance.

Workflakes is chosen to apply the changes, based on a hand written script. It would be a good change to have a self adapting Worklets on the system (evolving code) that uses previous gathered data and produces a script that fits its expected future data. The script could be generated and analyzed for predictability before actually being implemented or provided to Workflakes.

The use of ADL is being mentioned in the future work that the group is looking for, but the exact version of ADL or where they want to use it in the current model is not discussed.

2.7 Bandwidth Control.

Controlling the communication bandwidth in Microsoft Windows is possible and relatively simple. Starting from the basics of networking, a networking protocol depends on the ISO Open Systems Interconnection (OSI) model. The model determines the type of the connection between two nodes, and therefore sets the parameters of the connection. In the protocol stack, the transport layer governs the connection speed and data flow. This is the key control to the bandwidth of the connection.

A networking adaptor in MS Windows is designed to utilize the highest bandwidth possible from the connection. Yet, Windows provides the capability to manually adjust this parameter and choose the appropriate network speed.

The options provided in Windows 2000, NT and XP allow the administrator to adjust the connection speed to auto, 100Mbps full duplex, 100Mbps half duplex, 10Mbps full duplex, 10Mbps half duplex, or disable. This property can be controlled using Common Language Infrastructure (CLI) [33], and can be embedded in the code of the agent.

Reducing the bandwidth down to the allowed settings may not be the optimal answer to control the communications that the agent is trying to degrade in order to prevent the spread of the malware. Additionally, with the use of firewalls, all communication ports on the machine can be blocked, with the exception of the one port that the agent is using to communicate on the network. The choice of the port is a limited random selection, providing protection for the agent from being attacked by malware that might attempt to block its communication port.

The port control is managed by the firewall, as it prevents any communication from happening unless being certified. This will provide a bottle neck for the malware to communicate, given that by its nature, it will attempt to consume all the available bandwidth. Other self applications might have the capability to locate open ports to provide connectivity, and they will add more pressure on the port and on the malware.

2.8 Processes Control.

In the search for the processes control in the Microsoft Windows environment, a possible control paradigm can be applied from within MS Windows. The operating system provides a capability to control the priorities for executing the processes, which is extended into ending a process. Using the Common Language Infrastructure (CLI) commands, the agent could adjust the priority of a process, which gives the other *self* processes the privilege of taking most of the processor(s) time.

Another capability of the operating system is to assign priorities to processes across an entire array of processors. In this case, the agent should be aware that degrading the priority of the process will not prevent it from the chance to execute on another processor in multi processor systems. Therefore, it should assign all the targeted processes to a single processor, with high restrictions.

This solution may not be the best to handle the processes and their behaviors. In an article published in 1993, Wahbe et al. [34] discussed a framework for handling fault isolation, and a way to sandbox a process by assigning a separate processing area where the process is not allowed to write or jump outside it. Small and Seltzer [35] extend this concept by saying that even a read operation can change the priorities and privileges for resources' requests.

In this model, a better approach to handle novel processes that are still not classified as self components, is to apply a sandbox technique. Modern programing languages including C# and Java Corba can facilitate the control of processes by performing sandbox operation on them in a virtual processing environment and even manag the execution of these processes' threads [36]. This control can be performed in micro-kernel as added extensions, or can be done outside the kernel to prevent any violations to the kernel functionality in critical systems.

Windows *Runas* command can be used by the agent to provide simple control of the process. This can provide privilege control over the process, but it does not control the process activity and occupation of the processor time. User-level sandbox operations are an ideal mode to perform the control on the processes; this is performed by modifying the address space of all processes or logical protection domain to contain one or more shared pages of virtual addresses [37]. This method can control the privileges (write, read, and jump operations) which the process can perform; the extensions applied on the kernel can perform restrictions on the process piping for execution. This approach provides the ideal platform for controlling the processes.

In a review of our model, the best approach to control the *non-self* processes can be achieved by providing a sandbox environment for all the new processes. A sandbox environment provides a temporary storage for process output, and a virtual processing parameters and tools, such as registers and controllers. All the transactions that these processes perform can be rolled back if proven malicious, by simply dumping the temporary memory locations and files. If the application is registered by the administrator as *self*-component, then it can be allowed outside the sandbox to execute in the real environment by redirecting its pointers to the real processing environment. All the processes in the virtual space are prevented from exiting the controlled area until their signatures are extracted and identified by the agent or administrator as safe. This saves the time to recover from potential damage by containing the supposed effect in a throw away temporary storage, or for possible future changes, if needed.

III. METHODOLOGY

3.1 Basis of the Approach.

The primary concept this study is based upon is the approximation between the human immune system (HIS) and the artificial immune system (AIS). Therefore, the components of the AIS need to act in the same manner as the HIS components. Based on this approximation, any additions or modifications to the system can be derived from the HIS actual working (artificial life approach).

As discussed in section 2.1, the white cells in the HIS perform their function in a trial and error process. The cells try a certain protein compound to kill the antigen, and if it does not work, it will continue trying new compounds until the correct one is found, cloned and spread throughout the body. The agent in the AIS behaves in the same manner, using the antivirus and the signature extraction tools to try to identify the antigen and to remove it from the system. In this approximation, the agent acts like the B cells in the HIS.

Another basic concept used in this study is the reuse of the already provided components of the security systems such as antivirus software. This reuse will alleviate the cost of rebuilding the components by using applications tested previously and used in real systems.

Sandbox technique is another concept that provides protection from malware, by putting the new processes in a virtual processing area, to monitor them and prevent any malicious changes to the system. If the new process was certified by the administrator, it is then released to be executed in the real system and be allowed to utilize the system's resources.

3.2 The System's Architecture

Despite all the efforts in trying to reach better security, hackers are still able to pose a great threat to the computer systems with the help of the technology advancement. One of the main factors is *time*, through which the security systems' developers can react to a threat posed by the attacker. An analyst can look at the attack in an attempt to build an antidote, after which the system becomes immune from the same attack again. The nature of the beast is that viruses and the like have to strike first before the anti-viral is prescribed. This is similar to the HIS, where the disease has to first attack the body before it receives the medicine or treatment.

The architecture proposed here depends on the teamwork among the security applications; their work must be coordinated to avoid wasting time in identifying one another and resolving conflicts among them. The model is implemented on *distributed system* architecture (Figure 3), and the operating system has a multi processing capability to allow the agent to handle more than one process at one time.

In this architecture, the operating system environment is assumed to give capabilities and permissions for programs to call each other and execute code embedded within other containers, such as scripts in emails or mobile code in HTML containers. This is the worst case scenario for security, but it is the most spreading configuration among Internet users with machines using MS Windows. These settings must be disabled to prevent many intrusions; most average users do not know the severity of keeping such options enabled without proper monitoring.

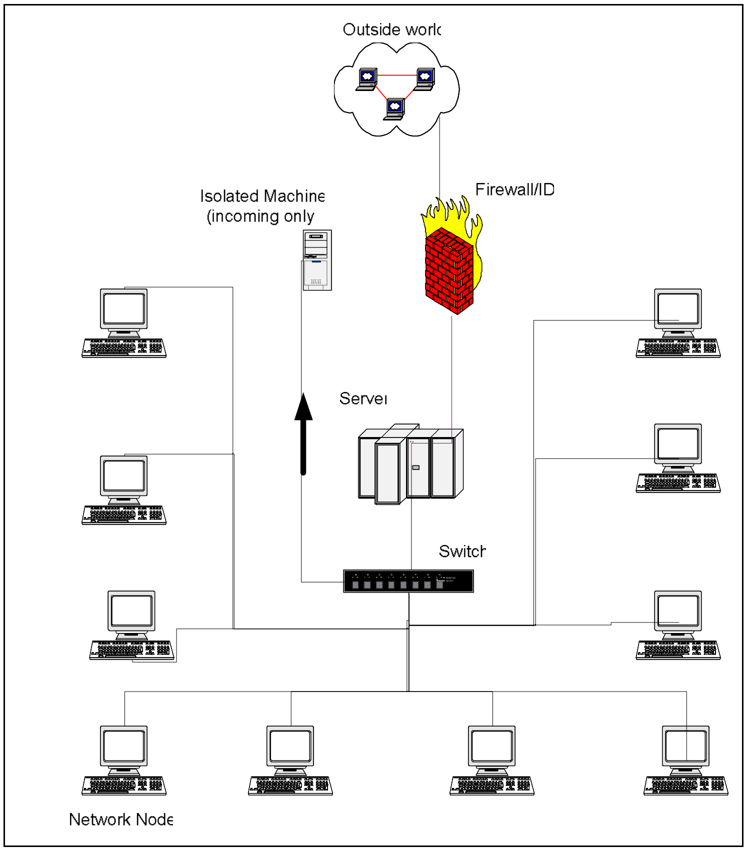


Figure 3. Structure of the Proposed Network Model.

Antivirus (AV) developers depend on code signature to identify viruses, at the same time virus developers strive to develop techniques to elude the virus scanning mechanism, such as using polymorphic viruses to camouflage copies of the virus. Polymorphic viruses change its virus signature (i.e. its binary pattern) every time it replicates, yet the core of the virus remains the same, and what is actually changing is the shell around the virus. In an attempt to partially solve this problem, *anomaly-based* intrusion detection (ID) and *behavioral-based* malware detection were introduced to help encounter unknown threats that have predictable behavior. In many experimental cases, these techniques proved effective with new threats, but more work needs to be done to improve them; mainly, in the field of adaptability to new anomalies. Recognition is improved by using techniques such as neural networks to recognize new patterns of the behavior to reproduce better generations of the recognizers and reach better optimization. Research in this field is still in progress and requires more confidence in the approach. In interest of “brevity”, the current state-of-the-art in this field is accepted as given.

3.3 Characteristics and design of the Agent.

A definition of a software agent is “a software entity which functions continuously and autonomously in a particular environment often inhabited by other agents and processes” [13]. Other definitions found in the literature portray the same concepts with different wording. In essence, for a program to be considered as an agent, it has the following characteristics [3]:

- Reactivity: the ability to selectively sense and act.
- Autonomy: goal-directedness, proactive and self-starting behavior.
- Collaborative behavior: can work in concert with other agents to achieve a common goal.
- Knowledge-level communication ability: the ability to communicate with persons and other agents with language more resembling humanlike “speech acts” than typical symbol-level program-to-program protocols.
- Inferential capability: can act on abstract task specification using prior knowledge of general goals and preferred methods to achieve flexibility; goes beyond the information given, and may have explicit models of self, user, situation, and/or other agents.
- Temporal continuity: persistence of identity and state over long periods of time.

• Personality: the capability of manifesting the attributes of a “believable” character such as emotion.

• Adaptivity: being able to learn and improve with experience.

• Mobility: being able to migrate in a self-directed way from one host platform to another.

It is worth mentioning the difference between an agent and a daemon, where a daemon is a program that runs continuously and exists for the purpose of handling periodic service requests that a computer system expects to receive. The daemon program forwards the requests to other programs (or processes) as appropriate. For an example, each server of pages on the Web has Hypertext Transfer Protocol daemon (HTTPD) that continually waits for requests to come in from Web clients and their users. The main difference between a daemon and an agent in this study is that a daemon can not make decisions on proper actions; it simply receives and carries out commands while the agent makes decisions and gives command to other applications.

An agent should be able to carry out activities in a flexible and intelligent manner; it should be responsive to changes in the environment without requiring constant human guidance or intervention. In this architecture, the agent intelligence is very primitive, the fact that it is simulating the B-cells which shows no intelligence but a mere systematic process. In future work of the agent, the behavioral monitors could be constructed to be able to adapt to new anomalies and train them to add to its knowledge, which will add intelligence characteristics to the agent. The agent should function continuously in an environment over a long period of time and be able to learn from its experience. Furthermore it inhabits an environment with other agents and processes.

In this application, the agent has a main goal to manage the components of the AIS, by *directing* their attention, and *managing* their reactions. For an example, the agent utilizes the AV to scan suspect processes, while the AV is in the status of in-the-fly scan, which requires the AV to scan the agent first to assure its validity. Afterwards the agent will direct the AV to activate a custom scan of the target. At which time, the agent should be inferential of the standard process of the AV. Otherwise a deadlock might occur where each task preserve the right to go first. To achieve coordination, the AV provider has to modify his product to allow the agent to assume control over the AV.

Another important task is the reproduction, which aids the agent in gaining an overall system control [8]. The agent cannot have the ability to know everything happening in the whole domain. However, it is task oriented, so it is able to provide complete control of its local domain. This requires the agent to have duplicates in every station in the domain to dominate locally after gathering the required information to function in their new environment. Agents should have a kind of *identification* such as encrypted identification keys, to provide validity and a correctness check and to avoid hoax agents. The regeneration of the agents should be done from a controlled station such as a server machine. This server provides the identification keys to the agents, and sends them to the stations that are requesting them.

If the agent engages a malware in one machine, then it triggers an alert to all other agents in the other machines in the domain. After the current suspected code is handled, the high alert situation is removed and the normal operational configuration is restored. Agents have a certain critical age defined by the administrator to assure the agent always have the latest anomaly algorithms. When the agent reaches its critical age it reports to the server to be terminated and a fresh copy of the latest agent is generated and released to inhabit the node.

The next task the agent should be able to handle is reading the signature of the malware. The entire AV products perform signature extraction on the code that they scan. Researchers in IBM describe and provide an automated mechanism that can extract the signature of a virus or any code [7]. In following comments on the IBM research site, the author discusses the new threats of macros and worms, and provides an alteration to the virus-signature extraction approach to handle the new threats. So an automated technique could be utilized within the agent to identify the malware signature, yet the technique offered refers to certain cases where the human interference is required, such as handling polymorphic code. In this architecture, polymorphic code is still in a sandbox, and the act of replicating is a sufficient condition to identify an anomaly.

Another working of the agent is the sandbox technique performed to deprive the malware of the freedom of movement, while providing a better environment for the AV to work with the code, and protecting the system from any actions the malware might perform. Sandbox in this case includes a *rollback* of any changes the malware performed, to avoid any changes that are not desirable if the code was identified as “harmful” afterwards (refer to [36] and [37] for details on this process). Those changes include any attempted outgoing communications, registry changes and any other parameters or saved data.

The agent is the cornerstone of this system. It monitors the behavior of the system, decides the proper action, and reports to the administrator. To perform the required actions, it must have a goal to achieve, which is to prevent any non-self code from executing freely. Another requirement for its success is the plan based on stimuli/reaction pairs [31]. The plan might actually consist of more than one sequence of actions and possible alternative actions to be followed throughout the agent life cycle.

To provide the agent with some feedback to help facilitate its workings, a knowledge base is provided. This acts as the memory for the agent, to distinguish self from non-self components. The agent monitors the environment by applying probes on the processes in sandboxes to provide knowledge of anomaly existence [32]. To provide the agent with reference to the boundaries that it needs to govern, a periodic count of the base running processes is generated by the agent. This information is provided by the probes to conduct a comparison and tell of the existence of newly added processes. The count contains the number of processes loaded in the safe and clean startup of the system. This provides the agent a reference to observe. When the administrator installs a new application, the agent requires the administrator to register the new application in the knowledge base. This updates the signature database (the agent performed signature extraction of the new application) by adding the new signature to its memory. The knowledge base contains all the signatures of the applications installed by the administrator.

The administrator is informed of the existence of the anomaly by the agent, and is informed of the system's behavior. The agent encourages the administrator's intervention, but can manage without it. If the agent becomes overwhelmed with new processes and could not handle them, then the administrator needs to terminate some of the processes in the sandboxes to allow the agent space to work, or inject the system with the proper antigen if it is available at that point. Frequent updates of the AV definitions are very helpful to the agent, to go through the shorter process of just cleaning, rather than the longer process of sandbox, investigate and clean.

3.4 Architecture workflow

Once a station is started, the station will request an agent from the server, this is like requesting an IP address from a DHCP server, or MS Windows checking for updates. The request includes the station identity encrypted by its private key and a time stamp. The station identity can be a combination of identifying characteristics, for an example its MAC address, the machine name, and a password. The server and the station identify each others request and reply using asymmetric cryptosystem. If the request is valid, the server will send an agent to the system with the server identity, and then the system verifies the agent's validity and executes it.

The agent starts it work by instructing the AV to perform a scan of the memory for known malware. Once the AV returns with a clean memory result, the agent runs a scan of the memory and extracts the signature of all the processes and puts them in a table, then checks each signature against a database of self processes. For each process, if a process is a "non-self", the agent will inform the administrator of its existence, puts it in a sandbox (which is a virtual processing environment) using the sandbox tools, and starts preventive action, which will be explained later.

When a process is loaded in memory, the agent sandbox it in a virtual environment to ensure that it does not perform any illegal operations or harmful acts. The agent identifies processes by extracting their signatures and then search for the signature in a database of known/trusted applications to avoid a sandbox operation of self components. In general, agents reside in the stations performing the B-cell job in the HIS by looking for any suspicious activity such as a process cloning or reproducing. The agent realizes such fact by using the behavioral monitoring tools which identify such anomalies.

When the process is loaded in memory, the agent directs the AV to scan it against known malware, then it is terminated and the administrator is informed. If the AV could not recognize it as malware, it will be allowed to execute in its sandbox until the administrator instructs the agent to allow it to execute outside the sandbox or terminate it.

If the administrator identifies the process as friendly, the agent logs it in its memory database as “self”. In the future, if the same process activates again it will not be put in a sandbox. If the administrator says that the process is unfamiliar, then the agent will perform *preventive action* which allows the process to continue executing in its sandbox under supervision of the anomaly detection tools. If the process at any time performs an anomaly, the agent will inform the administrator and terminate the process which includes dumping all the changes made in the virtual processing environment in a temporary file for the administrator to evaluate. This can be stored in a quarantined machine controlled and used only by administrators. The signature of the anomaly is added to the AV and ID database to be considered in the future as a malware. This database is accessed by all the agents on the network to update their AV and ID to identify the malicious process and terminate it (Figure 4).

If the administrator can not identify the process then it will remain in the sandbox to execute, providing it does not show any odd behavior. This will lead to two possible scenarios: the process will finish executing normally, which requires it to exit the sandbox, but all the changes are saved in temporary files for the administrator to evaluate if he wishes, then he can decide to commit these changes or just dump them. The other possibility is that the process will terminate due to the sandbox effect, such as running out of resources or not receiving responses, this will result in saving all the changes made in the virtual environment for future action by the administrator and the owner of the process to be notified of the reason of the termination.

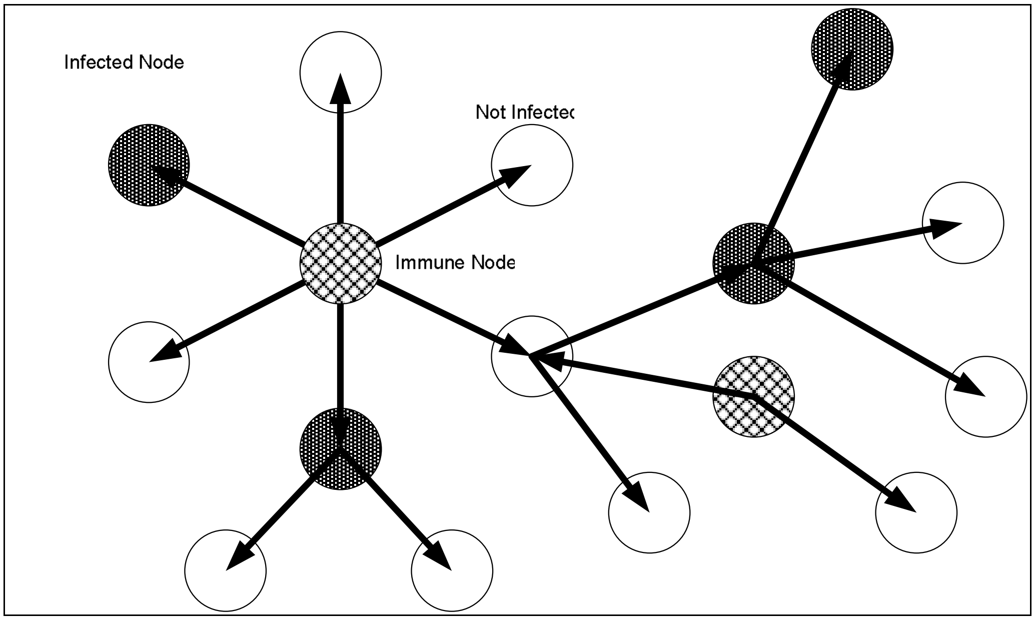


Figure 4. Immunity Propagation Among The Nodes.

If the agent detects an anomaly, the administrator is alerted of the specific action, and asked if he wishes to trust the process, if the administrator agrees to trust the process, then the agent will instruct the sandbox tools to allow the process to execute outside the sandbox. Otherwise the agent will terminate the process and throw away all the changes made by this process, informs the administrator, then the process' signature is added to the AV and ID database as a known malware. By doing so, the agent prevented the malware from making changes outside the sandbox, and helped keeping it in a controlled environment which concludes the preventive action of the agent work.

On the other frontiers, the ID handles the intrusion detection with two techniques, *anomaly detection* and *pattern recognition*. The first technique develops a sense/knowledge of the type of the transmitted data by gathering information based on a random collection of propagating data on the network. This can be classified as the training phase. Next is a random inspection and detection of the transmitted data on the network. This is considered the practical phase. If any abnormal data transfer occurs afterward, such transaction is put in a sandbox and the administrator is informed. Pattern recognition is used to check for certain patterns in the transactions, such as brute-force attacks or a code’s signature.

Another aspect is the *Firewall*, which manages the port connections coming into or going out of the network. The administrator has to manually set the policies to suit his vision for a secure gateway to his network. This part is an analogy to the parts of the human body; which allow communication with the external world. The firewall is running all the time with no ports left open unless the administrator allows for a good reason. Due to the fact that reconnaissance is important before any attack, log files must be monitored to try to prevent future attacks. *Auditing* the log files can help identify any bizarre attempts that may be an introduction to a more serious threat. The administrator should define two sets of policies, one for normal situations and another to be used when an anomaly is detected. Processes in a sandbox are not allowed to use the network, but providing limited access monitored by the ID can be allowed, as long as there are no anomalies in the system. If the agent declares an anomaly, the FW will apply its strict policy which prevents processes in sandboxes to use the network.

Appendix B provides a diagram of a visual description and an algorithm of the workflow of the agent. Additional figures are provided to visualization of the sandbox technique performed on new processes.

IV. CONCLUSIONS

4.1 Summary.

In chapter 1, this paper laid out the problem of malware and the difficulty in handling such code. The current approach to handle malware is to wait for it to hit then subscribe the cure. The purpose of the study was introduced, which is to use the human immune system (HIS) as a base for an approach to the artificial immune system (AIS), using agents to act as the B-cells and provide the artificial system with the self-healing capability against malicious software.

Chapter 2 provided a summary of the literature review done on the related subjects to this research, which provide better understanding of the biological model; it also included a look at the research on the AIS and the approaches to achieve a good comparison between the biological immune systems and the AIS. It is important to understand the threats that a system faces. Section 2.3 provides a brief look at the models of malware and their techniques. An approximation between the HIS components and the proposed AIS components was presented in section 2.4. The approximation provided an approach to design the agent to perform like the B-cells. Another idea used is the technique that the white cells follow when searching for the right protein compound that can be used against certain antigens. This concept inspires the trial and error approach that the agent uses to extract the signature of the malware to identify and remove it.

Self-healing systems were also reviewed in section 2.5 to get an understanding of the approaches used to provide the artificial systems with a self-healing capability without the need to take them off-line. A comparative study was provided to describe the strengths and weaknesses in the current research of self-healing. This comparison provides a novel approach to design a self-healing system against malware. The last two sections of chapter 2 described the possibility of controlling a process' speed and the communication bandwidth from within the agent's code.

Chapter 3 introduced the functionality of the system with section 3.1 describing the basis of the approach in the design of the architecture. The system architecture was introduced in the following section to present detailed functionality of the system. Section 3.3 provided the agent characteristics and design that enable it to control its environment and perform its job. Based on the provided architecture, the agent implementation is likely a straight forward operation that can be implemented within a proper environment of software development.

4.2 Conclusions.

Complete security is a dream, like every other dream we strive to make it a reality, even though we know it is almost impossible, but we have to try.

When this research started, the main concern was to provide the administrator with better control over the applications installed on his machine. With a closer look at the problems that face this approach, the research diverted toward finding a better way to handle malware in general with the use of smart code such as an agent. To provide a work frame for such an agent, the agent's living environment must be examined. That is where the approach of artificial life became a part of the research. Putting all these elements together provides a self-healing system, inspired by the human (biological) immune system, for healing from security “diseases”. Therefore, this research provides a bridge between the mere self-healing approaches and the security concerns.

So far, self-healing studies focus on components swapping and architecture reconfiguration. These studies do not consider the security aspect of the system, and do not discuss the existence of malware as a component in the system. If such malware exists, a current self-healing system would treat it as a valid component with a bug, and would simply relocate it, possibly increasing its threat. Table 2 provides a comparative description of four recent approaches to self-healing systems, introduced in the last workshop on self-healing systems 2002. In the table, different approaches are described for viewing the system components and handling suggested changes.

	Graph based reconfiguration	Model-based adaptation	DMonA architecture	KX approach
Ability to force reconfiguration online	Can be implemented	Can be implemented	Can be implemented	Can be implemented
Reusability of the self-healing components	Not applicable	Applicable. Adds overhead to communication channels	Applicable. DMonA and DiPS are independent applications.	Applicable. Components are independent software entities.
Dependability among self-healing components	Tightly bonded, failure of one component might cause the system to malfunction	Lightly bonded. Components can be changed and moved without affecting the system performance	Work separately, needs each other to complete the job. Location is irrelevant.	Totally independent. Could be used by other applications while not in use by the self-healing system
Handling running operations (in process transactions)	Running transactions will be lost. No temporary storage or pointer handling	Use of ADL provides control over the transaction’s pointers.	Data are encapsulated. Data pipelines are removed as a whole.	No discussed in the literature. Indicated possible with the use of XML smart schemes.

Rollback to previous configuration	Not provided. Memory configuration is not discussed.	Not provided. Configuration memory is not discussed.	Not included, but could be implemented.	Discussed in detail. Uses Workflakes with configuration memory.
Feedback on the new configuration	Not discussed in the current publications. Mentioned in future work.	Immediate evaluation of the system performance after the change is made.	No discussed. Monitors are available to evaluate the change.	Provides feedback to focus gauges and probes, to provide better reading of the system performance.
Location of the monitoring tools	Instances of the architecture. Provides better knowledge, but extra overhead.	External to the system components.	External to the system components. Parts of the monitoring applications.	External to the system components. Independent applications.
Reconfiguration strategy scripts	Handwritten scripts of possible configurations	Tuning the system components to gain better performance.	Strategy management application handles the changes in the configuration.	Uses Workflakes with scripts of possible configurations.
Use of descriptive languages	Not used.	Uses ADL.	DMonA application uses xADL.	Workflakes.
Feedback techniques on the changes	Not provided. Discussed in future work, using evolutionary computation.	No tools used, only test system performance.	Not included in the design.	Not provided. Probes and gauges could be used to provide feedback to the XML event schemes.
Malware handling capability	Does not recognize malware, may relocate it as a legitimate component. Will possibly dump the changes made, but will not terminate it. Lack of feedback on system performance may result in an infinite loop of changes exhausting the system's resources. Reusability of the system components may result in spread of the infection outside the current station and possibly throughout the domain.			

Table 2. Comparison Between Four Self-Healing Systems.

The components of the system are all available in the market or under development, except for the agent proposed here. Antivirus and intrusion detection applications are researched continuously, with new features and techniques added with every new release, modifying them to accept the agent control is possible and required to achieve the overall system’s goal. Sandbox techniques and tools have been used in testing and evaluating applications, with capability to log every action made to decide on either commit these actions or abandon them later. Anomaly detectors have been used in intrusion detection techniques for a long time, and antivirus applications have recently adopted this technique to detect polymorphic malware, but with limited success. Anomaly detectors still need to be improved, but with the introduction of neural networks, it is possible to achieve better results.

New processes do not always have to be malware, the number of false positives depend on the sensitivity of the agent which is governed by the anomaly detectors. Predicting these values can be realized only by implementing and testing the proposed architecture. The administrator attention is needed, to reduce the number of false positives, yet it is not required. Absence of the administrator will result in loosing great amount of the memory and system resources to the new processes in the sandboxes, which will result in overall system slow performance. The administrator can adjust the agent response to leave the decision to him, but that will require the administrator to devote a lot of his time monitoring and responding to the agent’s requests. If the users of the system utilize the same type of applications, the system will have a steady learning curve after all the standard applications have been executed at least once, this brings the number of new processes much lower, giving the administrator more free time.

The self-healing architecture as described in this paper can be implemented and will probably deliver the expected service. The success ratio can only be determined after the actual implementation, but all the components have sensitivity parameters that can be adjusted to decide the overall system sensitivity. One of these parameters is the administrator and his involvement in the system's operations.

The current research and development of sandbox tools and anomaly detectors means that certain parts of this architecture may need to be developed alongside the development of the architecture. This could be considered by some as a road block, but it could be a good derive to enhance and test such tools within a working system.

4.3 Model Weaknesses.

Applying security to a system means applying restrictions on the components of the system. This is noticed not only in computer systems, but also in real life situations. These restrictions can result in less efficiency, slowness and the possible lessening of functionality in the original system.

This fact applies to this model, as a huge tradeoff is expected between efficiency/speed and security. The agent will execute new processes in a sandbox which requires more space from the memory and more time from the I/O devices. The firewall will secure the communication gateways to prevent the spread of the malware and the ID will filter all the network transactions when a threat is seen. This is not necessarily pleasant for the users, but slow performance is better rather than the consequence of malware spread, paralyzing the whole system later.

Users who are executing the new processes will notice a very slow performance due to executing these processes in a sandbox. The process is living in a virtual environment, meaning that it is another layer over the operating system. The operations that are executed by that process are not committed until the administrator agrees on its safety. Here, if the administrator can not identify the process it might terminate due to the lack of resources in the sandbox environment.

In this model, if the agent notices a generation of processes with different signatures, it will sandbox them and try to monitor all of them. This means that the agent will be busy and will use a big chunk of the processor time and memory space. This generation of the processes can be either a malware regenerating with polymorphic technique, or it can be a simple compiler generating processes for a user writing a program and running it. All these processes are not signed in because the programmer is still debugging and fixing the code. This model will not handle this situation in a good manner, except by blocking all requests from that user; other solutions could be applied to fix this problem, depending on the nature of the system and the possible use of its services.

Administrator attention can be of great help in this architecture. In the human body, a kidney is not planted without the proper tests to make sure it has the correct signature. The same applies in this model; the administrator saves the agent a lot of trouble by defining a process as “self”, resulting in the agent saving the system’s time bypassing the analysis and monitoring of the new process

In many cases, a new malware is possibly not a threat to a system if the system users are cautious enough in their daily use. This means the malware will have an antivirus signature produced by the AV producers before the malware reaches the system. Therefore the administrator is required to stay up-to-date with the new virus signatures and AV updates. This will save a huge amount of time that the agent might have spent trying to handle the malware, when the cure is already available.

REFERENCES

[1] http://alan.kennedy.name/crohns/primer/immunsys.htm, The Human Immune System, accessed Dec 2003

[2] Behavior-Blocking Stops Unknown Malicious Code, Network Magazine, June 2002.

[3] J. M. Bradshaw, "An Introduction to Software Agents", http://agents.umbc.edu/introduction/01-Bradshaw.pdf, accessed Jan 2004.

[4] Computer Security Threats, http://www.caci.com/business/ia/threats.html, accessed Nov 2003.

[5] S Forrest, "Self/Non-self Discrimination in a computer", Proceedings of the IEEE symposium on research in security and privacy 1994.

[6] http://uhaweb.hartford.edu/BUGL/immune.htm, Immune System, accessed Dec 2003.

[7] "Automatic Extraction of Computer Virus Signatures", http://www.research.ibm.com/antivirus/SciPapers/Kephart/VB94/vb94.html, accessed Jan 2004.

[8] N. R. Jennings, M. Wooldridge, "Applications of Intelligent Agents", from "Agent Technology Foundations, Applications, and Markets", publisher: Springer-Verlag, 1st edition, 1998.

[9] K. Regina, A. Boukerche, J. Bosco, M. Notare, "Human Immune Anomaly and Misuse Based Detection for Computer System Operations: Part II", Proceedings of the International Parallel and Distributed Processing Symposium 2003, IEEE © 2003.

[10] M. Neal, J. Hunt, J. Timmis, "Augmenting an Artificial Immune Network", International Conference on Systems, Man, and Cybernetics, 1998, IEEE © 1998.

[11] M. Garetto, W. Gong, D. Towsley, "Modeling Malware Spreading Dynamics", 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, Volume: 3, 30 March-3 April 2003, IEEE © 2003.

[12] R. E. Marmelstein, D. A. Van Veldhuizen, G. B. Lamont, "A distributed architecture for an Adaptive computer virus immune system", International Conference on Systems, Man, and Cybernetics, 1998, Volume: 4 , 11-14 Oct. 1998 IEEE © 1998.

[13] Y. Shoham, "An Overview of Agent-oriented Programming". Source: Software agents, Pages: 271 – 290, Year of Publication: 1997, MIT publications.

[14] S.A. Taheri, G. Calva, "Imitating the Human Immune System Capabilities for Multi-agent Federation Formation", Proceedings of the 2001 IEEE International Symposium on Intelligent Control, 5-7 Sept. 2001 IEEE © 2001.

[15] Webopedia, http://www.webopedia.com/TERM/M/malware.html, last accessed Jan 8th 04.

[16] Z. Yanchao, Q. Xirong, W. Wendong, C. Shiduan, "An Immunity Based Model for Network Intrusion Detection", Proceedings of the International Conferences on Info-tech and Info-net 2001, Beijing, Volume: 5, 29 Oct.-1 Nov. 2001 IEEE © 2001.

[17] L. M. Bridwell, P. Tippett, "ICSA labs 7th annual computer virus prevalence survey 2001", http://www.trendmicro.com/NR/rdonlyres/C490C780-DF65-43FB-9629-9A6EE23B804E/2770/icsavps2001.pdf. Last accessed Mar 7 th 04

[18] M. van Oers, McAfee AVERT "Unix Shell Scripting Malware", http://www.net-security.org/, Posted on 7/26/2002; last accessed Mar 14th 04.

[19] E. M. Dashofy, A. Hoek, R. Taylor, "Towards Architecture-Based Self-Healing Systems", Workshop on Self-Healing Systems (WOSS) 02, Charleston SC, USA, ACM.

[20] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, N. Parlavantzas, "Reflection, Self-awarness and Self-Healing in OpenORB", Workshop on Self-Healing Systems (WOSS) 02 Charleston SC, USA, ACM.

[21] P. Oreizy, N. Medvidovic, R. N. Taylor, "Architecture-Based Runtime Software Evolution", April 1998, Proceedings of the 20th international conference on Software engineering, ACM.

[22] M. M. Gorlick, R. R. Razouk, "Using weaves for software construction and analysis", May 1991, Proceedings of the 13th international conference on Software engineering, ACM.

[23] J. Kramer, J. Magee, "Distributed Software Architectures", International Conference on Software Engineering (ICSE) 97 Boston MA, USA, 1997 ACM.

[24] M. Wermelinger, J. Luiz Fiadeiro, "A graph transformation approach to software architecture reconfiguration", 2000, Citeseer.

[25] M. Wermelinger, J. Luiz Fiadeiro, "Algebraic software architecture reconfiguration", ACM SIGSOFT Software Engineering Notes, Proceedings of the

7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, October 1999.

[26] M. Wermelinger, J. Luiz Fiadeiro, "A Graph Based Architectural (Re)configuration Language", European Software Engineering Conference (ESEC)/ Foundations of Software Engineering (FSE) 2001, Vienna, Austria, ACM 2001

[27] D. Garlan, B. Schmerl, "Model-based Adaptation for Self-Healing Systems", Workshop on Self-Healing Systems (WOSS) '02, Nov 18-19, 2002, Charleston, SC, USA. 2002 ACM

[28] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, P. Verbaeten, "Self-Adapting Concurrency: The DMonA Architecture", Workshop on Self-Healing Systems (WOSS) '02, Nov 1819, 2002 Charleston, SC, USA, ACM.

[29] G. Valetto, G. Kaiser, "A case study in software adaptation**",** Workshop on Self-Healing Systems (WOSS) '02, Nov 1819, 2002 Charleston, SC, USA, ACM.

[30] R. T. Monroe. "Capturing software architecture design expertise with Armani". Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, Oct. 1998.

[31] J. Bryson, "The Behavior-Oriented Design of Modular Agent Intelligence", http://www.cs.bath.ac.uk/~jjb/web/bod.html, Proceedings of Agent Technology and Software Engineering, 2002.

[32] N. Janssens, E. Steegmans, T. Holvoet, P. Verbaeten, "An Agent Design Method Promoting Separation Between Computation and Coordination", Symposium on Applied Computing (SAC) '04, March, 2004, Nicosia, Cyprus, ACM.

[33] Best Practices for Managing Applications with Process Control, Microsoft TechNet, http://www.microsoft.com/technet/prodtechnol/windows2000serv/deploy/prodspecs/proctrl.mspx, last accessed May 2004.

[34] R. Wahbe, S. Lucco, T. E. Anderson, S. L. Graham. "Efficient software-based fault isolation". 14th Symposium on Operating Systems Principles (SOSP). ACM Operating Systems Review, SIGOPS, volume 27 number 5.

[35] C. Small, M. Seltzer. "A comparison of OS extension technologies". USENIX (the Advanced Computing System Associations) Annual Technical Conference, winter 1996. Citeseer.

[36] S. Chan-Tin S. Reiss, "Sandboxing programs", http://www.cs.brown.edu/people/sct/sandboxing.programs.sct.pdf, last accessed June 2004.

[37] R. West, J. Gloudon, “User-Level Sandboxing: a Safe and Efficient Mechanism for Extensibility”, http://www.cs.bu.edu/techreports/pdf/2003-014-user-level-sandboxing.pdf, last accessed June 2004.

[38] P. L. Meintjes, A. G. Rodrigo, “Evolution of relative synonymous codon usage in Human Immunodeficiency Virus type-1”, Proceedings of the second conference on Asia-Pacific bioinformatics, ACM, 2004.

[39] P. K. Lala and B. Kiran Kumar, “Human Immune System Inspired Architecture for Self-Healing Digital” Systems, Proceedings of the International Symposium on Quality Electronic Design (ISQED.02), 2002 IEEE

[40] O. Nasaroui, F. Gonzalez, D. Dasgupta, “The fuzzy artificial immune system: motivations, basic concepts, and application to clustering and Web profiling”, Proceedings of the 2002 IEEE International Conference on Fuzzy Systems, 2002. FUZZ-IEEE'02.

Appendix A: Human Immune System Components

Figure 5. The Process By Which T Cells And B Cells Interact With Antigens [6].

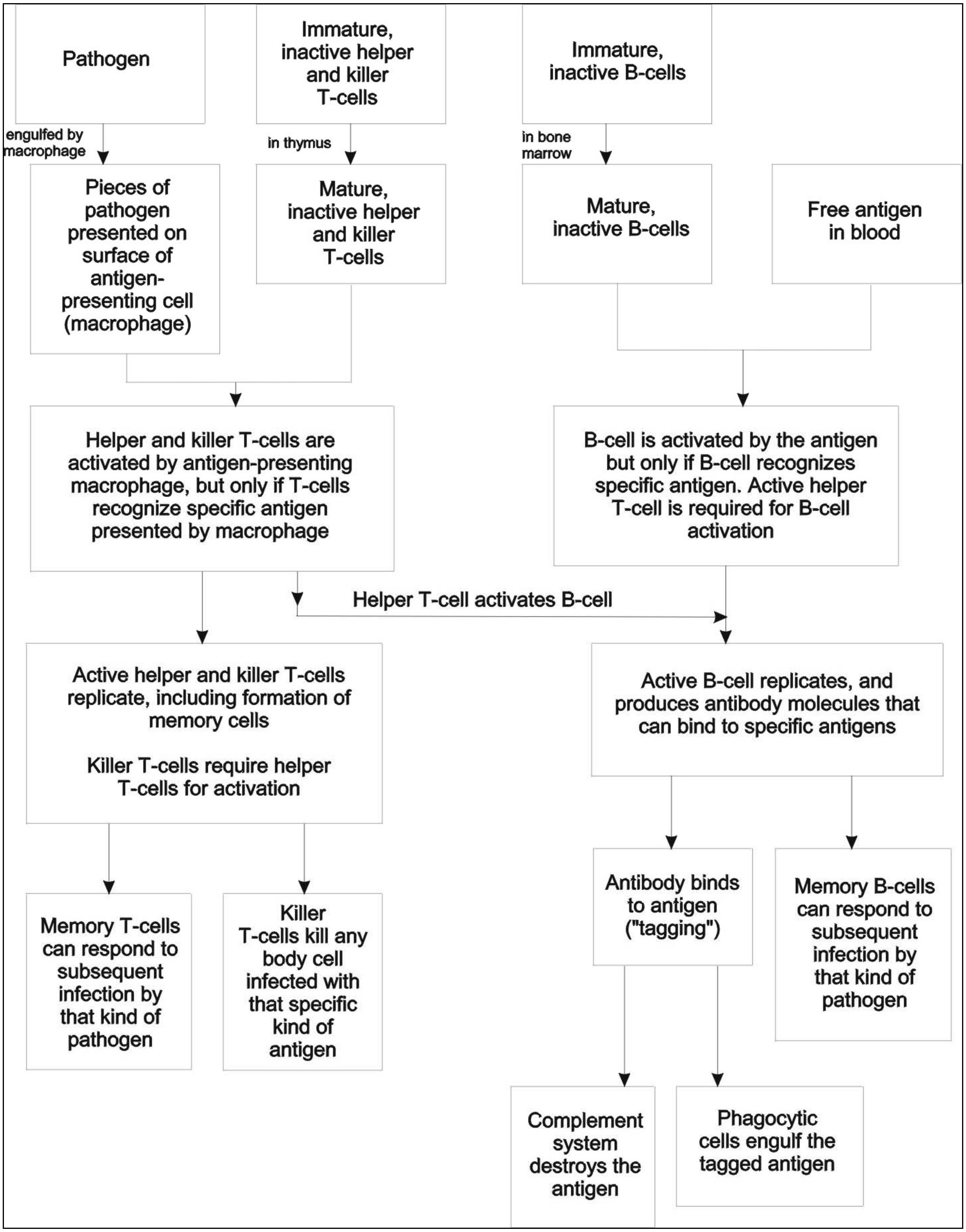
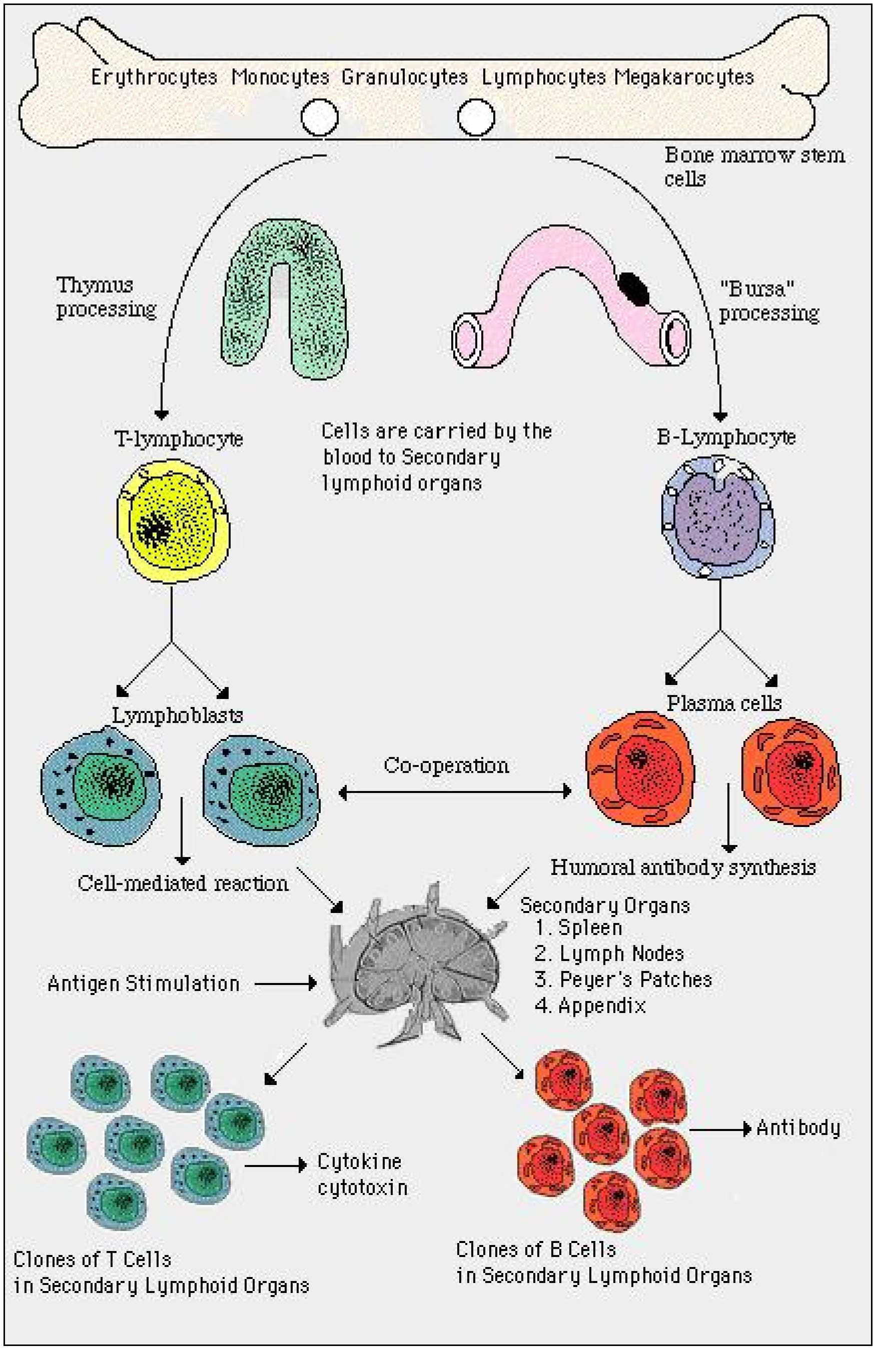

Pathogen
engulfed by macrophage
Pieces of pathogen presented on surface of antigen-presenting cell (macrophage)
Immature, inactive helper and killer T-cells
in thymus
Mature, inactive helper and killer T-cells
Immature, inactive B-cells
in bone marrow
Mature, inactive B-cells
Free antigen in blood
Helper and killer T-cells are activated by antigen-presenting macrophage, but only if T-cells recognize specific antigen presented by macrophage
B-cell is activated by the antigen but only if B-cell recognizes specific antigen. Active helper T-cell is required for B-cell activation
Helper T-cell activates B-cell
Active helper and killer T-cells replicate, including formation of memory cells
Killer T-cells require helper T-cells for activation
Active B-cell replicates, and produces antibody molecules that can bind to specific antigens
Memory T-cells can respond to subsequent infection by that kind of pathogen
Killer T-cells kill any body cell infected with that specific kind of antigen
Antibody binds to antigen ("tagging")
Memory B-cells can respond to subsequent infection by that kind of pathogen
Complement system destroys the antigen
Phagocytic cells engulf the tagged antigen

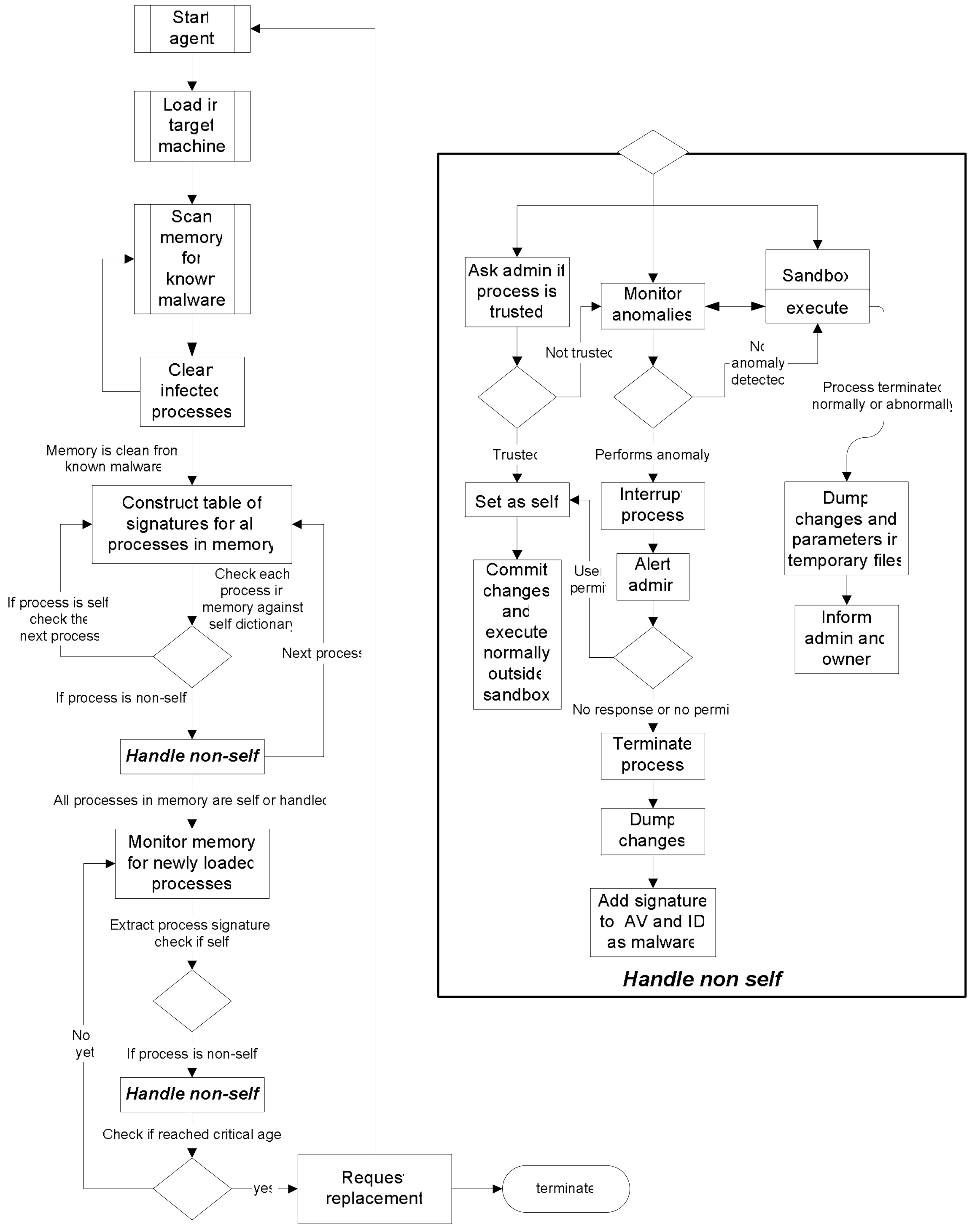
Figure 6. Graphical Representation of the Life Cycle of T Cells and B Cells and Their Interactions with Antigens.

From University of Hartford, Department of Mathematics, Epidemics and AIDS web page [6].



Appendix B: Agent Design

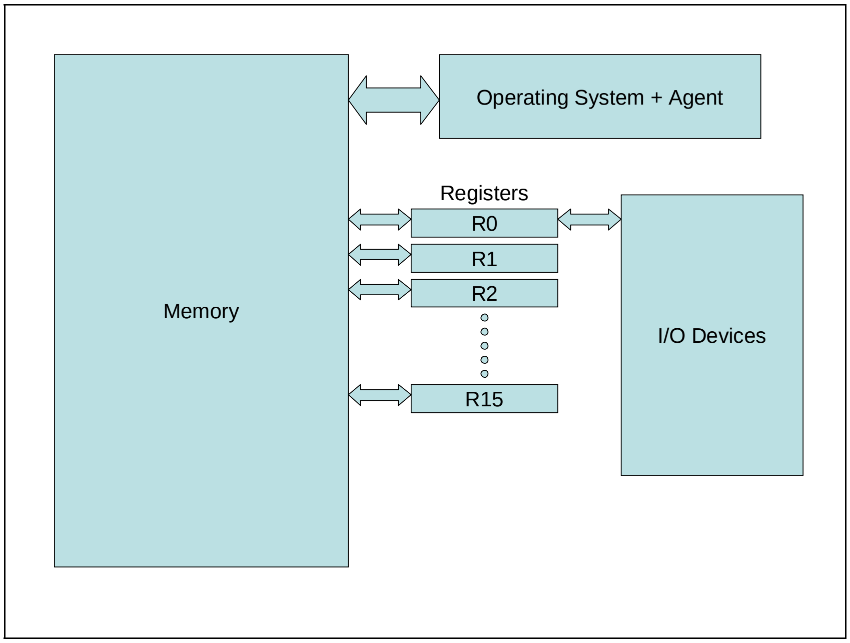
Figure 7. Agent Structure And Flow Diagram


Starl agent
Load ir target machine
Scan memory for knowr malware
Clear infectec processes
Memory is clean fron known malware
Construct table of signatures for al processes in memory
Check each process ir memory against self dictionary
If process is self check the next process
Next process
If process is non-self
Handle non-self
All processes in memory are self or handlec
Monitor memory for newly loadec processes
Extract process signature check if self
If process is non-self
No yet
Handle non-self
Check if reached critical age
yes
Reques replacement
terminate
Ask admin if process is trusted
Monitor anomalies
Sandbox
execute
Not trustec
No anomaly detectec
Process terminatec normally or abnormally
Trustec
Performs anomaly
Set as self
Interrupt process
Dump changes and parameters ir temporary files
Commit changes and execute normally outside sandbox
User permit
Alert admir
Inform admin anc owner
No response or no permi
Terminate process
Dump changes
Add signature to AV and ID as malware
Handle non self

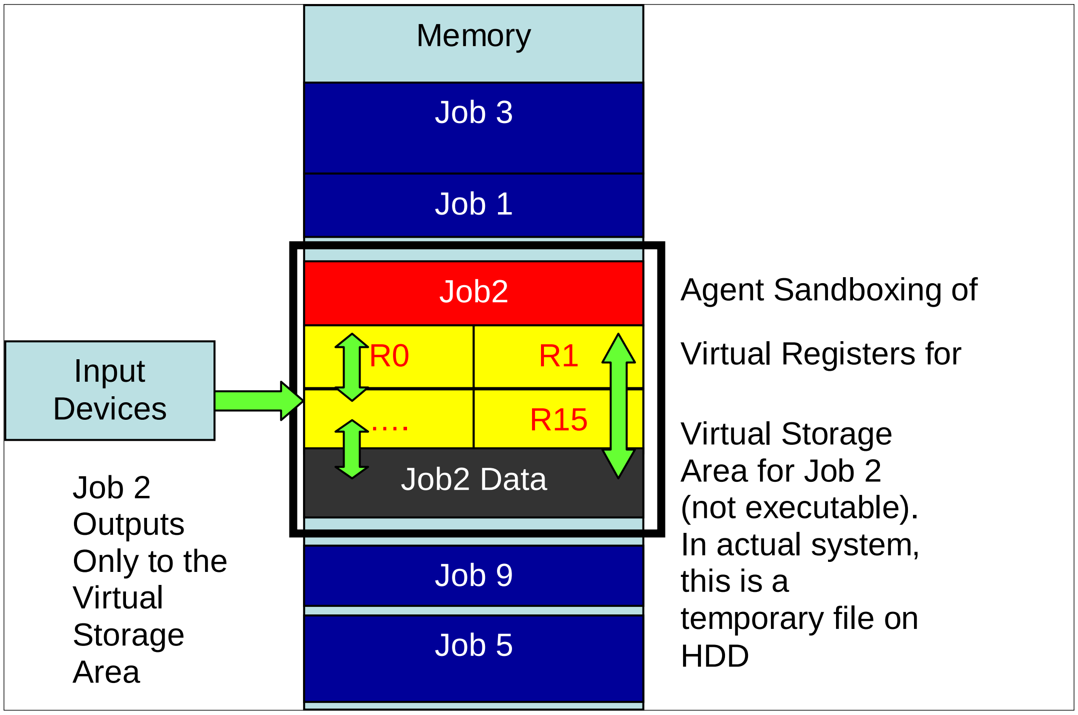
Algorithm 1: Agent algorithm.

```
initializeAgent (){
    requestAgent (timestamp, hashkey)
    Receive agent from server
    If valid hash key{ //run agent
         Agent()
    }
}//end AgentCreation
Agent(){
    Run antivirus to scan and clean system memory for known malware
    for each process in memory{
        signature=extractSignature(processID)
        Scan memory for processes according to “self” database entries
        If process is not in “self” database{
            Status=PreventiveAction(processID, signature)
            //log status
        }//end if
    }//end for
    // after assurance that all processes in memory are “self”
    while agentAge<criticalAge{
        Monitor all processes loaded in memory
        If loaded process not self{
            Run antivirus to scan if a known malware{
                if yes, terminate process and update log file
                else status=PreventiveAction(processID, signature)
                //log status
        }//end if
    }//end while
}//end ActiveAgent
string PreventiveAction(processID, signature){
    //Inform admin of process existence without interrupting its execution
    cobegin
        sandbox(processID)
        begin{
            print(Would you like to allow <process> to execute outside its
            sandbox? Y/N)
            read  reply
            if reply is yes{
                setSelf(processID, signature)
                return “self”
            }
        }end
        //use the anomaly detector to monitor the process behavior
        if process performs an anomaly
            status=alert(processID, signature)
    coend
    if process terminates normally or due to the sandbox effect{
        //such as out of memory, no response from admin…etc.
        Inform admin and owner of termination status
        Update log files
    }//end if
    return status
}//end PreventiveAction
string alert(processID, signature){
    Interrupt process
    Enforce firewall strict traffic policy
    print(<process> performed anomaly, terminate it? Y/N)
    read  reply
    Wait for predetermined period of time //admin defined, maybe zero
    If no response || Yes{
        Terminate process
        Save process’ temporary storage into an un-executable temp file
        Save process’ virtual processing area parameters in temp file.
        Update AV and ID with process signature
        Update log file with actions taken
        status= “malware”
    }//end if
    else{//admin identifies process as self
        setSelf(processID)
        status= “self”
    }//end else
    return status
}//end alert
setSelf(processID, signature){
    //use sandbox tools to redirect the process pointers to the real system components.
    Allow process to execute outside the sandbox
    //copy saved data to target files
    Commit changes made by the process to output devices
    //allow changes to memory and registers
    Commit changes made to the execution environment
    Add signature to “self” database
}//end setSelf
sandbox(processID){
    //using sandbox tools
    Create virtual processing area in memory (virtual memory and registers)
    Create temporary storage area for process output
    Execute process
}//end sandbox
string extractSignature(processID){
    Use signature extraction tool
    return signature
}//end extractSignature
```

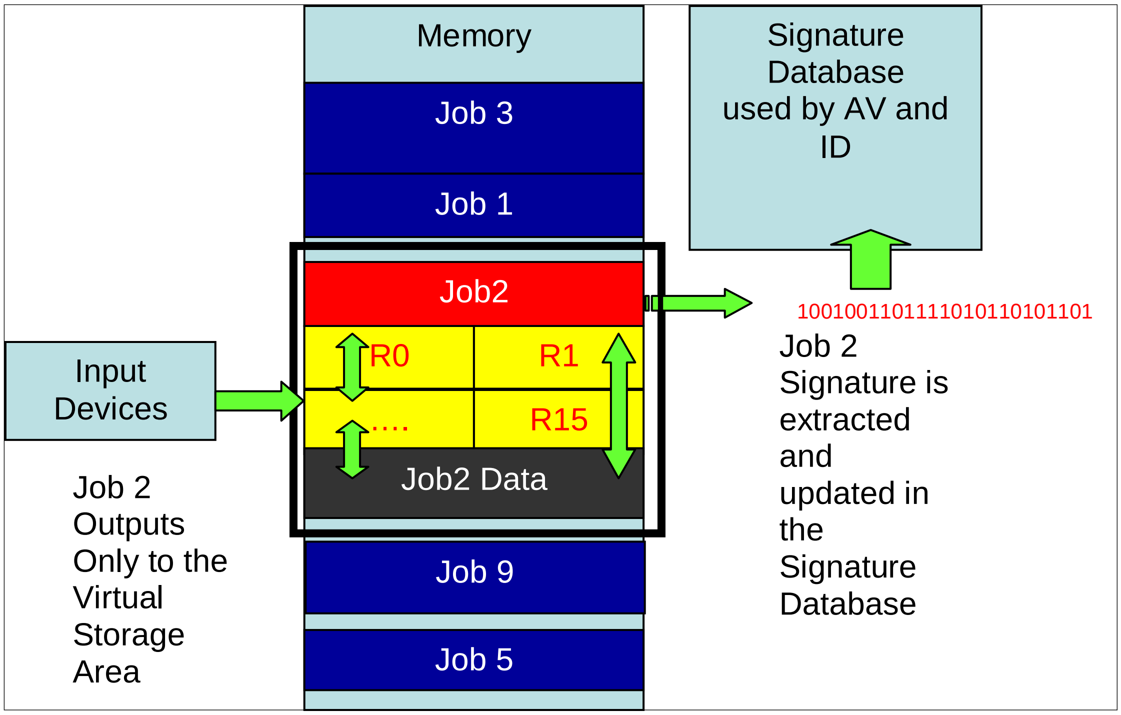
Figure 8. Sandbox Technique Sequence.



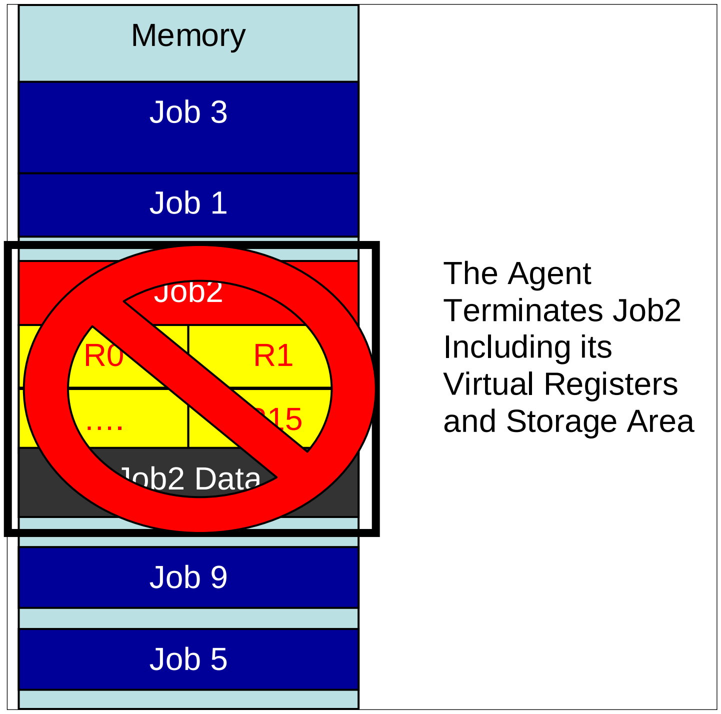
a. Part of the system hardware components


Memory
Job 3
Job 1
Job2
R0
R1
....
R15
Job2 Data
Job 9
Job 5
Input Devices
Job 2 Outputs Only to the Virtual Storage Area
Agent Sandboxing of
Virtual Registers for
Virtual Storage Area for Job 2 (not executable). In actual system, this is a temporary file on HDD

b. Job 2 is a “non-self” process, agent creates virtual processing environment for it to execute, with registers and temporary storage in memory or virtual hard drive.


Memory
Job 3
Job 1
Job2
R0
R1
….
R15
Job2 Data
Job 9
Job 5
Input Devices
Job 2 Outputs Only to the Virtual Storage Area
Signature Database used by AV and ID
1001001101111010110101101
Job 2 Signature is extracted and updated in the Signature Database

c. Job 2 performed an anomaly, the agent updated the AV and ID signature database with Job 2 signature for future reference.


Memory
Job 3
Job 1
Job2
R0
R1
….
R15
Job2 Data
Job 9
Job 5
The Agent Terminates Job2 Including its Virtual Registers and Storage Area

d. Job 2 is terminated and all the changes it made are abandoned along with its virtual processing environment

VITA

Bashar Barrishi
Candidate for the Degree of
Master of Science

Thesis: Modeling the Artificial Immune System to the Human Immune System with the use of Agents.

Major Field: Computer Science.

Biographical:

Personal Data: Born in Amman, Jordan, on November 27, 1972, the son of Salah and Hanan Barrishi.

Education: Graduated from Hussein College (High School), Amman, Jordan in May 1990; received Bachelor of Science degree in Computer Science and a minor in Military Science from Mutah University, Karak, Jordan, in June 1994. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in (December 2004).

Experience: Employed as a car mechanic during summers when at school. Was granted the silver and bronze medal prize for the Crown Prince Achievement when at school. Special Forces Captain and reconnaissance information officer at the Jordanian Armed Forces Special Operations Command. Employed by Oklahoma State University Library as systems assistant student. Employed by Oklahoma State University, Computer Science Department as a teaching assistant.

Professional Membership: Association for Computing Machinery (ACM), Phi Beta Delta (OSU) society.

Name: Bashar Barrishi Date of Degree: December 2004

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: MODELING THE ARTIFICIAL IMMUNE SYSTEM TO THE HUMAN IMMUNE SYSTEM WITH THE USE OF AGENTS.

Page in Study: 76 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: The purpose of this study is to provide a model and a work frame to approximate the artificial immune system to the human immune system with the use of agents to counter malicious software (malware). The artificial immune system components are commercial off-the-shelf products that are managed by the agent that coordinate and synchronize their activity. The behavior of the agent is a simulation of the B-cells in the Human Immune System in the encapsulation, analysis and digestion of the antigen.

Findings and Conclusions: The proposed architecture can be implemented in almost certainty based on the use of the commercial off-the-shelf products (COTS). The agent can be constructed to perform the required functionality with the help of the sandbox tools that provide the encapsulation. Anomaly detectors provide the knowledge of any process' action that is considered abnormal, hence, a possible malware. The Antivirus applications provide the digestion of the antigen, where known malware is handled directly, while unknown malware is analyzed by signature extraction, then handled by the antivirus. Other components such as intrusion detection (ID) applications perform the defenses at the entrances to the system (communication channels) and the firewall applications provide the prevention of the spread of the antigen and quarantining it in the infected node. The implementation of the model will provide a parallel self-healing system against antigens along side the applications and hardware self-healing systems.

ADVISOR'S APPROVAL: Blayne Mayfield