Intrusion Detection by Random Dispersion and Voting on Redundant Web Server Operations

by

Dennis Ohsuk Kwon

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2002

Certified by..... [illegible] ....
William Weinstein
Senior Researcher, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by ....
Howard E. Shrobe
Principal Research Scientist, MIT Artificial Intelligence Laboratory
Thesis Supervisor

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

Intrusion Detection by Random Dispersion and Voting on Redundant Web Server Operations

by

Dennis Ohsuk Kwon

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Until now, conventional approaches to the problem of computer security and intrusion tolerance have either tried to block intrusions altogether, or have tried to detect an intrusion in progress and stop it before the execution of malicious code could damage the system or cause it to send corrupted data back to the client. The goal of this thesis is to explore the question of whether voting, in conjunction with several key concepts from the study of fault-tolerant computing - namely masking, redundancy, and dispersion - can be effectively implemented and used to confront the issues of detecting and handling such abnormalities within the system. Such a mechanism would effectively provide a powerful tool for any high-security system where it could be used to catch and eliminate the majority of all intrusions before they were able to cause substantial damage to the system.

There are a number of subgoals that pertain to the issue of voting. The most significant are those of syntactic equivalence and tagging. Respectively, these deal with the issues of determining the true equivalence of two objects to be voted on, and "marking" multiple redundant copies of a single transaction such that they can be associated at a later time. Both of these subgoals must be thoroughly examined in order to design the optimal voting system.

The results of this research were tested in a simulation environment. A series of intrusions were then run on the voting system to measure its performance. The outcome of these tests and any gains in intrusion tolerance were documented accordingly.

Thesis Supervisor: William Weinstein
Title: Senior Researcher, Charles Stark Draper Laboratory

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist, MIT Artificial Intelligence Laboratory

Acknowledgments

I would like to thank my parents, Young-Dae and Eun-Jae Kwon, for their constant love and support throughout the past twenty-two years of my life. I would also like to thank God for giving me the strength to succeed when I needed it most. Thanks also to all of my friends and my sister, Jeannie, without whom I would have been unable to make it through this year. Last, but not least, I would like to thank Bill Weinstein and Dr. Howard Shrobe for their continued guidance and support as I prepared this thesis.

This thesis was prepared at the Charles Stark Draper Laboratory, Inc., under Contract #N66001-00-C-8061, with DARPA/ISO.

Publication of this thesis does not constitute approval by Draper or DARPA/ISO of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

(Author's signature)

[This page intentionally left blank]

Contents

1 Introduction **10**

1.1 Problem 10

1.2 Objective 11

2 Background **14**

2.1 Computer Attacks and Anti-Intrusion 14

2.1.1 Common Attacks 15

2.2 Anti-Intrusion Systems 17

2.2.1 Intrusion Prevention 19

2.2.2 Intrusion Preemption 19

2.2.3 Intrusion Deterrence 19

2.2.4 Intrusion Deflection 21

2.2.5 Intrusion Detection 21

2.3 Related Work 23

2.3.1 A Scalable Intrusion-Tolerant Architecture (SITAR) 24

2.3.2 A Decentralized Voting Algorithm for Increasing Dependability in Distributed Systems (TB-DVA) 25

2.3.3 New Methods of Intrusion Detection using Control-Loop Measurement 27

2.3.4 Intrusion Tolerance by Unpredictable Adaptation (ITUA) . . 28

2.3.5 Building Adaptive and Agile Applications Using Intrusion Detection and Response 29

3 Design Approach **32**

3.1 KARMA System Overview 32

3.1.1 Assumptions 32

3.1.2 Phase One KARMA System 33

3.1.3 Phase Two KARMA System 34

3.1.4 Required Components 36

3.2 Architectural Analysis 39

3.2.1 Narrowing The Parameters 40

3.3 Detailed Design Analysis 42

3.3.1 Non-Voting 42

3.3.2 Voting 44

3.3.3 Voting at M_{in} 44

3.3.4 Voting at M_{out} 56

3.3.5 Voting at G_{out} 57

3.3.6 Redundant Databases 61

3.3.7 Single Database 62

3.4 Design Conclusions 63

3.4.1 Final KARMA System 63

4 Implementation **65**

4.1 KARMA-1 65

4.1.1 Gateway 66

4.1.2 Configuration Manager 67

4.2 KARMA-2 67

4.2.1 Libasync - An Asynchronous Socket Library 68

4.2.2 Gateway 69

4.2.3 Transaction Mediator 76

5 Results **83**

5.1 Attack Types 83

5.2 False Alarms 84

5.3 Test Results 84
5.4 Response Times 86

6 Conclusions **89**

7 Future Work **91**
7.1 Performance Enhancements 91
7.2 Security Enhancements 92
7.3 General Enhancements 93

A Source Code **94**
A.1 Gateway 94
A.1.1 webproxy.h 94
A.1.2 webproxy.C 99
A.2 Mediator 110
A.2.1 mediator.h 110
A.2.2 mediator.C 113
A.3 G_{out} Voting Equivalence Algorithm 120

List of Figures

3-1 The Phase One KARMA System 34
3-2 The Phase Two KARMA System 35
3-3 Final KARMA System 63

4-1 Gateway Block Diagram 74
4-2 Mediator One 77
4-3 Mediator Two 77
4-4 Mediator Block Diagram 80

List of Tables

3.1 Brief Architectural Analysis 43
3.2 SQL Variable Types 51

4.1 Phase One KARMA Configuration 66
4.2 Phase Two KARMA Configuration 68

5.1 Test Results 85
5.2 KARMA Response Times 87

Chapter 1

Introduction

1.1 Problem

With the recent rise of the Internet and the World Wide Web (WWW), the number of people using computers and Internet services has grown at an alarming rate. According to an independent study performed by NPR in 1999, more than 8 in 10 Americans under the age of 60 had used a computer either at home or at work. In addition, more than half (52%) of all Americans under the age of 60 had access to e-mail or the Internet in their own homes. Since 1999, the number of computer users has grown exponentially [16]. As quickly as the number of people using the Internet has grown, so has the number of attacks made on Internet-based hosts.

Earlier this year, the Federal Bureau of Investigation reported that "nearly 90% of United States businesses and government agencies suffered hacker attacks within the past year." Furthermore, it was reported that the damage caused by Internet attacks cost US companies nearly $455 million last year, up by $125 million from the previous year [9]. With numbers like these, it goes without saying that the study of Internet security and intrusion prevention is an exciting and rapidly growing area of research.

Until now, conventional approaches to the problem of computer security and intrusion tolerance have either tried to block intrusions altogether, or have tried to detect and intrusion in progress and stop it before the execution of malicious code could damage the system or cause it to send corrupted data back to a user. The detection of such an intrusion has required that systems effectively look for anomalous patterns in behavior, incorrect or unexpected values for parameters, or attempts to perform unauthorized actions. The biggest problem with these methods of detection is that they are often limited to looking only for known patterns of attack. However, more sophisticated attacks that do not employ blatantly abnormal behaviors, but instead string together a series of seemingly benign transactions to collectively compromise a system, could pass through the system undetected. As a result, attempts have been made to come up with a more precise definition of what is considered to be abnormal behavior in a secure system. To date, such techniques have led to a large number of false alarms that have made it difficult to work with such intrusion tolerant systems. This thesis will present a new approach to intrusion tolerance called KARMA (Kinetic Application of Redundancy to Mitigate Attacks). KARMA confronts the issue of detecting abnormalities by adapting several key concepts from the study of fault-tolerant computing - namely masking, redundancy and dispersion. Furthermore, it can be shown that this new approach significantly improves the level of protection the system provides against intrusion and compromise.

1.2 Objective

The KARMA system utilizes a number of redundant components to detect anomalous behavior on the network. The system consists of a number of Internet servers residing on a network behind a trusted gateway. Each of these servers acts as a mirror, providing access to a variety of web content. In addition, the servers communicate with a database through a trusted mediator by utilizing a set of carefully written web scripts.

To an external user, the KARMA system looks like a single web server. The trusted gateway acts as a firewall between the internal servers and the outside world. Once a user has connected to the gateway, any request that is made is replicated and then randomly dispersed to a number of the internal servers (>3 to allow for majority voting). Each of these servers then processes the request and does one of the following:

1. If the request was for a dynamic page requiring database access - the web server forwards the database request to the trusted mediator which then compares its incoming requests from the various web servers and executes the query if they are found to match by some relative standard. The result is then returned to the corresponding web servers, where step 2 is then performed.

2. If the request was for a static page, or a dynamic page not requiring database access - a response is generated and sent back to the gateway. The gateway then compares the various responses for its original replicated requests and returns a response if they match. If the responses are found to differ by some relative standard, an appropriate error response is returned.

The KARMA system provides intrusion tolerance for "service" sites that support multiple external clients with web-based access to a large amount of information, database, and application services. It relies on a small number of trusted components to protect the functionality of a large number of untrusted components – untrusted in the sense that they are not subject to testing and analysis for security beyond that provided by the manufacturer, nor are any security-enhancements made to any of the existing operating systems or server applications. The objective of this system is the protect the confidentiality and integrity of the site's information in the presence of stealthy attacks – unknown to us – seeking to take advantage of vulnerabilities in the untrusted components of our system.

The goal of my research is to explore the questions of associating and voting on redundant web server operations. Following this introduction, the second chapter presents background information on the study of computer attacks and intrusion tolerance/detection. Chapter 3 gives an analysis of the various architectural/operational alternatives for the KARMA system leading up to a final design motivation. The fourth chapter then describes in greater detail the implementation of the KARMA system. Chapter 5 discusses some of the tests that were performed on the system and both the successes and failures of the KARMA system in handling them. Finally, Chapter 6 gives some conclusions drawn from the performance of KARMA, followed by a seventh chapter on future work to be done.

Chapter 2

Background

"A 'computer attack' may deny access to a system or compromise the confidentiality or integrity of data on that system. In the most threatening kind of attack, an intruder's privileges on a system are elevated beyond the norm." [11]

An intrusion detection system is one that attempts to detect attacks before they occur, thus preventing any harm to the networks or hosts that it is protecting. There are two types of intrusion detection systems: network-based and host-based. A network-based IDS analyzes traffic flowing through a network and attempts to identify attacks. Similarly, a host-based IDS monitors and examines the configuration of a system to effectively identify whether an attack has taken place. Each approach has its own set of advantages and disadvantages. The KARMA system utilizes both network and host-based components to reap the rewards of both approaches.

2.1 Computer Attacks and Anti-Intrusion

A computer attack is defined as "a sequence of related actions by a malicious adversary whose goal is to violate some stated or implied policy regarding appropriate use of a computer or network." (mcdonald 2001) The purpose of these attacks is varied and can range from the benign to the serious. An example of a benign attack would be a user attempting to explore a secure network to see what resources are available. An example of a more serious attack might involve an individual attempting to access privileged information, perform unauthorized operations on the system or bring down the system altogether [17].

Of the people who are attempting to compromise a system, there are three primary classes of attackers [17]:

- Masquerader - An individual who is not authorized to use the computer, and who penetrates a system's access controls to exploit a legitimate user's account.
- Misfeasor - A legitimate user who accesses data, programs or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges.
- Clandestine user - An individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection altogether.

2.1.1 Common Attacks

In nearly all cases of computer attacks, an attacker must attain a level of control that is beyond that which is specified by the system's security policy. The following are some of the most commonly used techniques for compromising a system or gaining illicit access to a system [11]:

- Password stealing - This is probably one of the most common methods used to compromise a system. There are two methods of stealing passwords. The more technical method is known as password sniffing. To sniff a password, an intruder observes the traffic flowing across a network and attempts to catch passwords being sent "in the clear" (without encryption) from a client to a server on the network. The less technical method is simply to steal a password. Some techniques an intruder might use to steal a password are looking through the trash to check if a user wrote it down somewhere, calling up a legitimate user and pretending to be a system administrator in need of their password, or less subtly, threatening a user to give up their password.

- Exploiting software bugs/misconfigurations - Often there are a number of known vulnerabilities in commonly used software that can be exploited granting a knowledgeable intruder an easy way into a system. In most cases, these vulnerabilities will leave the intruder with "root" or "super-user" privileges. Once inside, the intruder can then start a shell and begin executing custom programs of his/her choosing. Similarly, misconfigurations in software can leave open huge windows of opportunity for potential intruders. For example, if a windows administrator were to forget to disable guest access, they might then allow an intruder to easily gain access to their system and proceed to attack it from the inside.

- Physical access - This is the case in which someone might actually have access to the physical machine. If the machine were not networked this would pose less of a threat, in that it would only provide access to a single machine. However, if the machine were on an unprotected (trusted) network, it would be possible for the intruder to compromise every single machine with no trouble at all. In addition to compromising the network, a single physically accessible machine could also be compromised in a number of ways. The machine could be rebooted to bypass the password-entry mechanism. For example, in the default Linux installation, it is possible to login in single-user mode by setting a single startup flag, effectively providing the physical user with "super-user" privileges. Alternatively, if the machine contained large amounts of highly classified data the hard drives/storage media could be removed and read directly from another machine.

- Session stealing - Session stealing is the process by which an intruder would steal a connection from an already authenticated user. This could happen in several different ways. The intruder might flood the user's machine with packets (denial-of-service attack) thus taking their machine "offline," and then act as an impostor by stealing the original user's IP address. Another more common method of stealing sessions occurs in a system in which multiple users may share a single machine (i.e. public libraries, campus clusters, etc.). In this case, an intruder may be able to utilize cookie files that have been saved locally by another user. Cookie files are often used by websites to retain login information for authentication. Clearly, this would pose a serious threat if the website contained highly personal or sensitive information.

Obviously, this list does not encompass the entire range of possible attacks that can be used to compromise a system. Furthermore, as computers continue to get faster and more powerful, intruders continue to think of more and more ingenious ways to break into systems and bypass detection. Hence, an effective intrusion detection system must be designed to detect some, if not all, of these known attacks and yet also be flexible enough to catch those attacks that do not fall into these well-known categories.

2.2 Anti-Intrusion Systems

Initial efforts to combat intrusion turned to a method of intrusion prevention that essentially required system administrators to employ various techniques to design and configure a system such that they could effectively block all possible intrusions. However, it was quickly realized that this method would prove to be quite costly and would inevitably fail due to the sheer number of attacks that was growing in number every day. As a result, the idea of intrusion detection was born. Early intrusion detection systems were based on the theory that intrusions could be effectively detected by observing anomalous behavior on a network. An anomaly was defined as any behavior that was atypical of a user's historical behavior (as was documented by an audit log). It was thought that this audit mechanism would enable system administrators to catch not only intruders attempting to misuse the privileges of existing users, but also legitimate users attempting to perform actions beyond the scope of their access permissions.

Early experimentation showed that the auditing approach was indeed effective for keeping track of users' behavior patterns. Thus, greater emphasis began to be placed on new and interesting ways to apply techniques for audit trailing to the improvement of computer security. Eventually, this led to the development of various commercial intrusion detection systems “including SRI's IDES and NIDES, Haystack Laboratory Inc.'s Haystack and Stalker, and the Air Force's Distributed Intrusion Detection System (DIDS).”

Though much has been learned from the research of intrusion detection systems, it is important to also present some of the less recognized alternatives that have played a valuable role in the development of computer and Internet security as a whole [17].

There are a number of approaches that have been used to counter computer attacks. Of these, some of the most common approaches are [11]:

- Prevention - precludes or severely handicaps the likelihood of a particular intrusion's success.
- Preemption - strikes offensively against likely threat agents prior to an intrusion attempt to lessen the likelihood of a particular intrusion occurring later.
- Deterrence - deters the initiation or continuation of an intrusion attempt by increasing the necessary effort for an attack to succeed, increasing the risk associated with the attack, and/or devaluing the perceived gain that would come with success.
- Deflection - leads an intruder to believe that he has succeeded in an intrusion attempt, whereas instead he has been attracted or shunted off to where harm is minimized.
- Detection - discriminates intrusion attempts and intrusion preparation from normal activity and alerts the authorities.
- Countermeasures - actively and autonomously counter an intrusion as it is being attempted.

2.2.1 Intrusion Prevention

Intrusion prevention attempts to "preclude or severely handicap the likelihood of a particular intrusion's success." Essentially, this requires the administrator of a system to build and configure the system such that no security holes are left open and no possible vulnerabilities exist to be exploited by a potential intruder. Clearly this is an extremely high and virtually unattainable goal given the fallibility of the people who design and build such software. However, if such a level of prevention could be achieved, the security of the system would likely be unmatched in that no other anti-intrusion mechanisms would be required since it would be perfectly protected and there would exist no means by which to misuse the system [11].

2.2.2 Intrusion Preemption

Intrusion preemption techniques attempt to stop intrusion one step earlier than intrusion prevention. Some typical methods for doing so involve educating a system's users so as to inform them of proper routines for securing their computers, passwords, etc. In addition, laws could be passed to prevent systems from being exposed to compromising situations. Furthermore, if at any time, a regular user is observed to be demonstrating anomalous behavior action might be taken against that user to prevent them from performing potentially harmful actions from within the trusted network. These are just a few examples of the intrusion preemption mechanisms that are used today [11].

2.2.3 Intrusion Deterrence

Intrusion Deterrence techniques seek to make it appear to a potential intruder that breaking into a system is more trouble than it is worth. This could be done in a number of ways. The most common techniques are camouflage, warnings, paranoia, and obstacles [11].

- Camouflage - is the process whereby a system may appear from the outside to be less important than it actually is. This can be done effectively by masking the identity of the various components within a system. For example, if the system consisted of multiple servers protected by an external gateway, the gateway could fool an outside user into thinking that only one machine existed at the given external IP address. Even something as simple as renaming could be used to camouflage the true identity of a highly-valuable resource within a network (i.e. naming a database cinfo may prove to be less attractive to a potential hacker than one named CREDIT_CARD_INFO). In this latter case though, one could clearly see how the protocol might inconvenience some of the users of the system in that the names would not be as descriptive or informative, thus making the system much less intuitive to a new user.

- Warnings - some systems may seek to dissuade a user from breaking in by issuing multiple warnings to an attacker as they attempt to perform a compromising series of actions on the system. However, this method may often prove to be counterproductive as some attackers may be further egged on by the warnings for the exact opposite reasons for camouflaging. Attackers may see the warnings as a sign that something very important lies behind the "closed doors" to the system.

- Paranoia - is an attempt to scare potential attackers away by creating the impression that the system is very well guarded and monitored. This can be either true or false, but it is up to the attacker to decide whether they are willing to take the risk of getting caught.

- Obstacles - are put in place to augment the amount of time and effort needed to compromise the system. By creating dummy accounts, or by employing multiple levels of security, a system administrator may effectively be able to dissuade an attacker from expending the time needed to gain control of the system. Like the camouflage technique, it can easily be seen how this method might provide increased security at the expense of convenience for it's legitimate users. This is because the added obstacles could potentially create additional security protocols that would turn seemingly basic operations into annoying, complicated routines.

These are just a few of the techniques that might be employed in a system where the attacker is assumed to have a limit to their persistence or desire to attain access to the system.

2.2.4 Intrusion Deflection

Another commonly used method of anti-intrusion is intrusion deflection. This technique essentially tricks an attacker into thinking that they have successfully broken into a system only to find that they have been following a trail that leads nowhere. A honeypot is one example of an intrusion deflection system that is widely used today. The purpose of a honeypot is twofold. First, it acts as an alternate target to deflect attacks on the primary system. Secondly, it serves as a means of tracking and learning the actions of a potential attacker without putting any valuable information in harm's way [11].

2.2.5 Intrusion Detection

In William Stalling's book entitled "Network and Internetwork Security," he states that, "inevitably, the best intrusion prevention system will fail. A system's second line of defense is intrusion detection, and this has been the focus of much research in recent years."

The primary motivation for intrusion detection systems is that they enable a system to detect an attack and expel the malicious user before any irreversible damage can take place within the system. Even if an attacker does successfully manage to get inside a system, by expediently detecting the anomalous behavior, the amount of damage done can likely be mitigated.

As was stated earlier, intrusion detection is largely based on the assumption that one can distinguish, on the basis of audit records and behavior patterns, between a legitimate user and someone who has hacked into the system and is posing falsely as a real user. There are a number of techniques that are widely used today to detect such anomalies in behavior. The two primary categories of intrusion detection are [11]:

1. Statistical Anomaly Detection - As the name implies, this method refers to the long term collection of statistical data regarding the typical behavior patterns of a user. The system can then compare any given user's behavior with the gathered data to determine to a relative degree of confidence whether the user is an impostor. There are two main categories of statistical anomaly detection:

    - Threshold Detection - requires the system to keep a count of specific user actions performed within a given period of time. If a user's actions ever exceed or drop below the typical number of occurrences for a particular user, then an intrusion may be signaled.

    - Profile-based - this is very similar to threshold detection. However, rather than keeping count of a particular event and comparing all subsequent action counts to the collected data, a profile-based system attempts to keep track of multiple actions. Essentially, the system generates a long-term portfolio of a user's actions. Each of the actions that comprise a portfolio can then be ranked with a certain level of importance. This provides a more accurate method for comparison in that the effect of a legitimate user's occasional anomalous behavior could be diminished by other "more normal" behavior. However, if the user was indeed an attacker, it is highly likely that more than one of the actions in the profile would differ from the typical behavior of the given user, resulting in an overwhelming difference in behavior and this signaling an intrusion.

2. Rule-Based Detection - rather than detecting anomalous behavior by comparing user statistics, rule-based detection might detect an intruder on the basis of a set of rules that can be applied to the operations being performed on a system.

    - Anomaly Detection - rule-based anomaly detection is quite similar to statistical anomaly detection. The primary difference is that the rule-based approach compares a current user's actions against a set of rules for behavior (derived from audit records of a user's historical behavior). Each successive action can be compared against the set of rules to determine whether it falls into a previously defined pattern of behavior.

- Penetration Identification - This method is quite different from the anomaly detection routines in that it does not rely so much on historical patterns of behavior. Rather, it collects data on known attacks (by both technical and non-technical means, i.e. interviews with system administrators, developers, etc.). With this data, a set of rules is then generated through which any subsequent actions can be filtered to check for malicious patterns of behavior.

The previous two categories describe techniques that have focused primarily on host-based systems that need protection only for themselves. However a more interesting area of research, is that of network-intrusion detection, which is responsible for detecting attacks not only on a single host, but every host on the network.

Until recently, much of the work on intrusion detection was done solely for single host systems. However, the rise of the Internet and the prevalence of local area networks not only in businesses but also in homes, has provided motivation for a distributed, multiple-host, network-based intrusion detection system. The primary difficulty with network-based IDS is that since the IDS is not located on a single server, the audit records must be handled differently. Furthermore, since the distributed IDS might not have access to the low-level audit resources that a single-hosted detection system might have, a distributed intrusion detection system will often consist of multiple host-based components working together to form the entire anti-intrusion system.

2.3 Related Work

There has been a great deal of research conducted on the study of intrusion detection systems. In the following examples, a number of projects that are currently being developed in the area of intrusion tolerance will be presented. In addition, the advantages and disadvantages of each of the systems and the various differences between them will be discussed in greater detail. Lastly, an attempt will be made to illustrate some of the similarities between each of these projects and the proposed KARMA system.

2.3.1 A Scalable Intrusion-Tolerant Architecture (SITAR)

The Scalable Intrusion-Tolerant Architecture (SITAR) Project is currently being researched at Duke University and involves the development of "a scalable intrusion-tolerant architecture for distributed services in a network environment." The SITAR project attempts to succeed where many others have failed by *preventing* general intrusion attacks rather than by simply *detecting* an attack in progress. In the past, there has been very limited success in dealing with such direct approaches to intrusion tolerance. However, the SITAR project differs from its predecessors in that it does not focus on the intrusion attacks themselves, but on the functions and services that require protection – i.e. to be made intrusion tolerant. So rather than trying to detect intrusions on the system as a whole, the SITAR project attempts to monitor and detect intrusions on only those events that pose a threat to the services under examination. Hence, the system is able to effectively prevent intrusions to those services that are most essential to the performance of the system. In addition to examining only a generic set of attacks, the SITAR project improves upon current intrusion prevention systems by applying many of the basic techniques of fault tolerance, e.g. redundancy and diversity, to insure that the system stays dependable and predictable [20].

The most significant disadvantage of the SITAR project lies in the fact that it is an intrusion prevention system. Unlike many of its predecessors, the likelihood of success in preventing intrusions is much higher since the scope of attacks that need to be dealt with is much smaller. However, by virtue of the fact that it must be able to successfully identify any possible attack (in the smaller subset) in order to prevent a compromise, it is highly unlikely that such a system can be implemented with complete assurance of success.

2.3.2 A Decentralized Voting Algorithm for Increasing Dependability in Distributed Systems (TB-DVA)

This project, being developed at the Air Force Research Laboratory, addresses the problem of coordinating voting in a distributed system securely, so as to withstand malicious attacks. Conventionally, most distributed systems have utilized the same voting protocols to achieve fault-tolerance. One of the most common protocols, the 2-phase commit protocol requires the replicated voters to independently determine the majority rather than relying on a central source to tally the results. Once the final result has been calculated, one voter is chosen at random to commit their result to be passed on to the user. This method is widely used in developing fault-tolerant open distributed systems. However, the committal phase in this protocol opens the doors to a number of security holes [6].

The most obvious flaw in the 2-phase commit protocol occurs when a malicious attacker is able to compromise a committing voter. Then, the attacker can effectively control what result the user sees regardless of what result the other voters return. Though there have been a number of attempts made to modify the protocol such that it can be secured, most of these have been without success. This new solution, utilizes a timed-buffer distributed voting algorithm (TB-DVA), to effectively solve this problem.

The timed-buffer distributed voting algorithm (TB-DVA), involves a number of voters and an interface. Unlike the 2-phase commit protocol that requires each voter to vote independently and then one voter to return the result, TB-DVA, allows all voters to participate in a distributed manner in the final voting process. It does this via the following algorithm: [6]

Each voter performs the following steps:

1. If no other voter has committed an answer to the interface module, the voter

does so with its vote. It then skips the remaining steps.

2. If another voter has already committed, the voter compares its result with the committed result.

3. If the vote agrees, it does nothing. Otherwise, if the vote disagrees, it broadcasts its dissenting vote to all users.

4. Once all voters have had a chance to compare their results to the committed value, the voter analyzes all of the dissenting votes to see if a new majority exists.

5. If no majority exists, then the voter does nothing.

6. If a new majority exists, the voter commits this new result and returns to Step one.

The interface module performs the following steps:

1. Once a commit is received, the result is stored in a buffer and a timer is started. The timer is set to allow all voters to check the committed value and also have time to dissent if necessary.

2. If a new commit is received before the timer runs out, the new result is written over the old one in the buffer and the timer is restarted.

3. If no commit occurs before the timer runs out, then the interface module simply returns the value in its buffer to the user.

The TB-DVA improves greatly upon the 2-phase commit protocol by essentially enabling the voters to return a result correctly even if a number of the voters have already been compromised successfully. This works because it is neither a single voter, nor the centralized voter (in this case the interface module) that is responsible for coming up with the final result. Rather the result comes from a set of timed periods of communication between the interface and the distributed voters. Only if a majority of the voters has been compromised, would an intruder be able to dominate the result [6].

One disadvantage of the TB-DVA system is that it could possibly slow the performance and increase the complexity of the system. This is due to the fact that multiple cycles of voting must be completed before a final result can be returned.

2.3.3 New Methods of Intrusion Detection using Control-Loop Measurement

At the Fourth Technology for Information Security Conference in 1996, three researchers from Houston, Texas presented a novel idea for intrusion detection using control-loop measurement. The Control-Loop IDS attempts to apply concepts from digital signal processing (DSP) and control theory in electrical engineering. "Within this theory, a control system compares observations of a system's state with desired states to generate corrections intended to steer the system being controlled toward the desired state." [1]

Essentially, it can be shown that the traffic flowing through a network can be modeled as a digital signal, which can then be processed using modern statistical methods – "time-series data is collected, filtered, correlated, and analyzed for many purposes including event detection. The recognition and characterization of computer network protocols has been among the applications successfully handled by DSP." This finding, in conjunction with the control theory in electrical engineering, would allow researchers to quantify discrepancies in network behavior, thus providing them with the ability to distinguish between legitimate users and intruders [1]

The advantage of this new system is that it builds upon a tried and tested theory from electrical engineering (control theory) in addition to utilizing well-known concepts from digital signal processing. Hence, if it can be shown that the control-loop measurements can correctly distinguish between legitimate users and intruders to a relative degree of certainty, then this new method would provide a very enticing alternative to many of the newer intrusion detection systems that are in use today.

One disadvantage of this system is that it attempts to detect suspicious behavior based solely on patterns revealed by the control-loop theory in electrical engineering. It differs from typical intrusion detection systems in that it does not statistically collect historical data on pattern behaviors of ordinary users. However, this is a feature that could likely be integrated into the system with relative ease.

2.3.4 Intrusion Tolerance by Unpredictable Adaptation (ITUA)

"The purpose of the Intrusion Tolerance by Unpredictable Adaptation (ITUA) project is to develop a middleware based intrusion tolerance solution that would help applications survive certain kinds of attacks." The ITUA project proposes to do this by developing its own intrusion model of attacks that it will provide protection against. Attacks that are protected are called "covered" attacks, and those that are not are called "uncovered." The key concept that makes the ITUA project so interesting is that, it doesn't necessarily have to stop the attacker from ever getting inside. Rather, it may allow for an attacker to get inside the system, however, given that the system has several layers of defense, it is assumed that in most cases, the system will be able to catch the attack before it is able to reach any of the core services. Once the attack has been detected, adaptation is crucial because the system must not only be able to adapt to the attack so as to recover, but it must also do so in a manner that is unpredictable. Otherwise, an attacker might be able to predict the action that would be taken and simply prevent that action from ever taking place [23].

The advantage to this solution lies in its element of surprise. Rather than trying to detect and prevent an attack as most IDSs would, the ITUA project attempts to outsmart an intruder who is himself trying to outsmart the system. Hence, if the system manages to be unpredictable enough to befuddle even the most adept hacker, then it has a significant advantage in that it will be able to adapt and rebuild the system while an intruder's unsuspecting attack is thwarted.

Unfortunately, there are a few major assumptions upon which the ITUA project relies that could render the project unsuccessful. The most important of these assumptions is that "only staged attacks are 'covered."' A staged attack is one in which the intruder attacks certain outer security layers before attempting to break into the core services. This could be for any number of reasons – i.e. to learn more about the system's security configuration, or to try and compromise several less important machines to later launch an organized attack on a more significant one. This assumption is necessary because it gives the system time to respond and adapt to an attack. The problem with this assumption is that if an intruder were to launch an arbitrary attack on the system that did not adhere to this idea of "staged attacks," it might have a good chance at compromising the system before it had time to react. This would effectively render the ITUA system useless since, by the time an attack was detected, it would almost certainly be too late to reverse the attack's ravaging effects [23].

2.3.5 Building Adaptive and Agile Applications Using Intrusion Detection and Response

This project attempts to improve upon existing IDS technology by increasing the amount of communication performed between the various IDS components protecting a system and also between the IDS components and the applications which they seek to protect. This is made possible by using a custom framework called the Quality Objects framework, or QuO for short. The following are just a few of the advantages of using the QuO framework for intrusion detection [22]:

- "The development of intrusion-and-security aware applications" - QuO provides a framework upon which security aware applications can be built. As a result, each application becomes a mini-IDS of sorts and can aid in the intrusion detection process by reporting back suspicious activity to the actual IDS. Furthermore, QuO includes support for inserting custom audit mechanisms that would effectively allow applications to track not only the performance of security, but also attacks as they occur. This information would then prove useful in improving the system to prevent future attacks.
- "The development of survivable applications" - Since each application is "secu-

rity aware" it would be easier to integrate survivability into these applications. Hence, if a service were compromised, actions could be taken to mitigate the effects of the attack, to contain its effects, or eliminate them altogether by adapting the system to the malicious changes.

- "Integration and interfacing of multiple IDSs at the application level" - It is clearly possible for intrusions to be detected at a certain level. Furthermore there are a number of intrusion detection systems that are both commercially and freely available. However, it is the tendency for each of these systems to be strongest in detecting only certain kinds of attacks. Hence, if multiple intrusion detection systems were able to work in conjunction with one another, the number of intrusions that could be caught would increase significantly. The QuO framework allows different IDSs to communicate with each other, thus improving the range of "covered" attacks and resulting in a more secure system.

- "Integration of IDSs and other resource managers" - By integrating the IDSs and other resource managers it would be possibly to tighten the security within a system. This is because each of the resource managers (i.e. security policy manager, dependability manager) operates at a lower system level whereas the IDS might perform certain higher-level tasks. Thus, if an intrusion were detected at the high-level of an IDS, it might be able to effectively warn the lower-level resource manager allowing it to respond in such a way as to prevent the attack from causing further damage.

The functionality provided by the QuO framework is this system's clearest advantage. By tying each of the components in the system (IDS, applications) so closely together, it creates a tighter-knit system overall and thus makes it more difficult for compromises to be made since, in essence, every suspicious action is being watched and communicated across the board [22].

The clearest disadvantage of this project is that it adds significantly to its complexity in that all of the applications and IDSs must work in conjunction with one another. Furthermore since all of the applications and IDSs are based on the QuO framework, an improper design could lead to a set of very difficult upgrade and maintenance procedures. Likewise, since every component works within the QuO framework and is so tightly bound to the rest of the system, if an IDS or application were successfully compromised, it might be much easier for this attack to wreak further havoc within the system.

The above examples are just a few of the projects currently being conducted in the field of anti-intrusion and intrusion tolerance. The next chapter on design, will contain a detailed analysis of some of the similarities and differences between the KARMA system and many of the IDSs that are either in use or in development today. In addition, an argument will be made to describe why KARMA would be effective as an intrusion detection system and illustrate some of its potential applications.

Chapter 3

Design Approach

3.1 KARMA System Overview

The KARMA system provides intrusion tolerance for "service" sites that support multiple external clients with web-based access to a large amount of information, and database/application services. It relies on a small number of trusted components to protect the functionality of a large number of untrusted components. To be untrusted simply means that these components should not be subject to any testing or analysis for security beyond that which is specified by the manufacturer. Furthermore, no security-enhancements are made to any of the existing operating systems or server applications that these untrusted components will use. In other words, all of the untrusted components will be using commercial-off-the-shelf, i.e. COTS, operating systems and server applications. The objective of the KARMA system is to protect the confidentiality and integrity of such a "service" site's information in the presence of stealthy attacks seeking to take advantage of vulnerabilities in the untrusted components of our system.

3.1.1 Assumptions

The KARMA system offers protection for "service" site under the assumption that attacks on the integrity or confidentiality of data require that an adversary gain a certain level of control on the relevant system, not merely take it out of service (i.e. the system is not designed to stop denial-of-service attacks). As a result, to execute a particular exploit, the attacker needs to know the details of the operating system and server software for the specific system that they are trying to divert. Currently, there are a number of widely available tools in existence that could be used to probe the platform over the network to determine what OS, architecture and web server are being used on the target system. However, one of the strengths of the KARMA system, that will later be discussed in greater detail, is its ability to mask the identities of its untrusted components.

The second assumption is similar to the "staged" attack assumption proposed in the ITUA project. In order to attack the integrity or confidentiality of data on the "service" site, it is assumed that an intruder must not only gain a certain level of control on the relevant system, but must also be able to maintain a connection with that system once it has been compromised in order to pass along further instructions to complete the attack. This is to say that an attack cannot be completed in a single step, but rather a typical attack will consist of a set of sequential operations that must occur in order on a single untrusted component. This assumption allows the system to effectively divert, or at the very least delay an attack by not performing all operations on a single untrusted component.

3.1.2 Phase One KARMA System

The KARMA development effort was conducted in two phases. The initial phase of the system is shown in Figure-3-1.

In the phase one KARMA system (KARMA-1), a trusted gateway disperses incoming requests to multiple untrusted components, called origin servers, providing web access to a large amount of information, and database/application services. In this phase, using only dispersion, the system's primary mechanism for defense is to "confuse" a potential attacker sufficiently by scattering all incoming transactions so that he/she will be unable to predict the system's behavior and exploit a known vulnerability for one of the untrusted components. Essentially, this is an application of

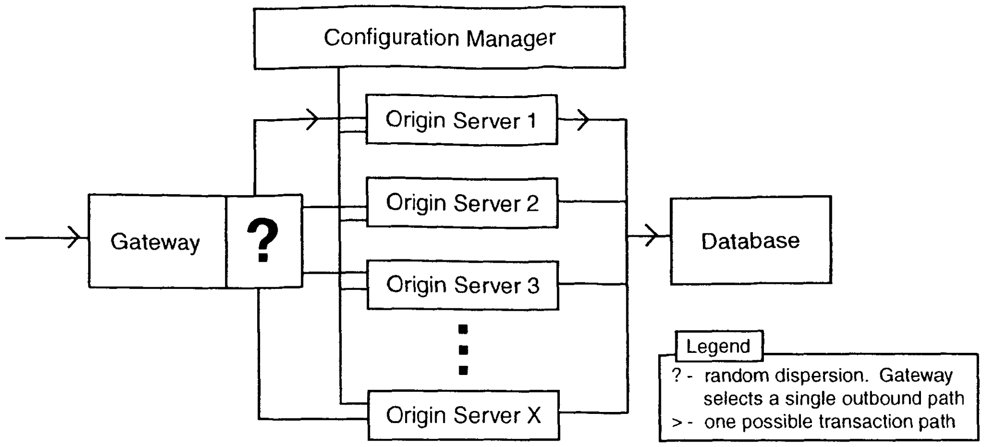


Figure 3-1: The Phase One KARMA System

the "staged attack" assumption. By effectively dispersing all incoming transactions such that no two are consecutively sent to the same origin server, it is believed that the system can thwart an attack by creating a discontinuity in the flow of commands to a single machine. In the case that an attacker is able to successfully compromise an origin server, the KARMA-1 system employs a second level of defense. This secondary defense is a network-based intrusion detection component known as the configuration manager. Although this component is itself network-based, it relies upon a number of host-based modules, residing on each of the individual origin servers, to report back to it with important host-level information. For example, a host-based module might be responsible for monitoring the set of scripts that are provided by an origin server. If any of these scripts is modified, it can then report this change back to the configuration manager.

3.1.3 Phase Two KARMA System

The second phase KARMA system (KARMA-2) builds upon the KARMA-1 system by inheriting a number of important concepts from the field of fault-tolerant computing – masking, redundancy and dispersion. These concepts are defined below:

- Masking - "Fault masking is a structural redundancy technique that completely masks faults within a set of redundant modules. A number of identical modules execute the same functions, and their outputs are voted to remove errors created

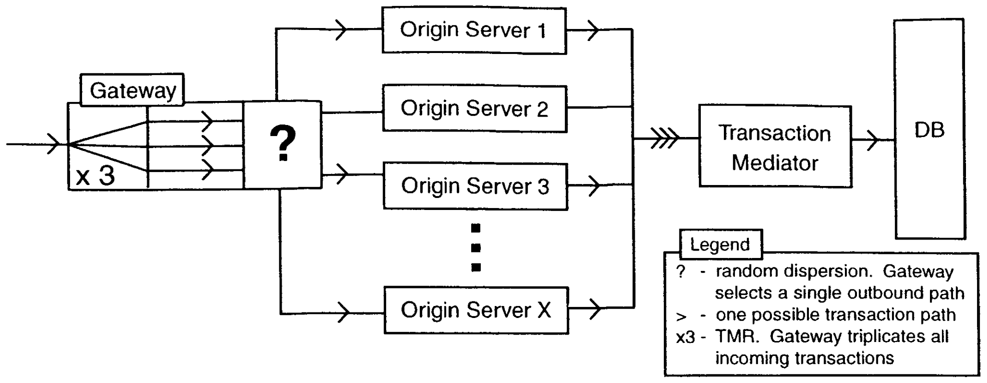

Gateway
x 3
?
Origin Server 1
Origin Server 2
Origin Server 3
Origin Server X
Transaction Mediator
DB
Legend
? - random dispersion. Gateway selects a single outbound path
> - one possible transaction path
x3 - TMR. Gateway triplicates all incoming transactions

Figure 3-2: The Phase Two KARMA System

by a faulty module. Triple module redundancy (TMR) is a commonly used form of fault masking in which the circuitry is triplicated and voted." [14]

- Dispersion - refers to the distribution of tasks to enhance the performance of a system. In the KARMA system, dispersion refers to the way that incoming transactions are sent to the various origin servers to improve the system's ability to detect and survive an attack.

- Redundancy - a technique in which multiple copies of information, data, and application services are maintained. The advantage of using redundancy is that it eliminates a single failure point for the system in regard to intrusion. In order to gain control of the entire system, each redundant copy, or at least a majority, must be compromised.

The KARMA-2 system explores the use of redundancy, masking and dispersion to enhance its ability to detect intrusions in real-time. Like the phase one system, the trusted gateway is responsible for taking incoming transactions and dispersing them to multiple origin servers at a time. However, instead of dispersing the original transaction to just a single server, the KARMA-2 system triplicates each of its incoming requests. Each copy of the original transactions is then dispersed to one of the origin servers.

As each transaction is received by an origin server, it may encounter a script that requires access to the database. If the action being performed is a database query, this query will then be forwarded to a transaction mediator that is responsible for handling the queries from all of the collective origin servers. Hence, a single incoming transaction at the gateway would result in a set of triplicated web requests that, given the nature of the requests, might in turn cause three separate database queries to be sent to the trusted transaction mediator.

Upon receiving all three of the triplicated database queries, the transaction mediator can then initiate a round of voting on the queries to determine whether the query should be passed on to the database or an error response should be sent back to the origin servers. If voting passes and the query is forwarded to the database server, once the database server has processed the query and returned its results, the mediator can then opt to vote on the results as well. If both voting stages are completed successfully, each of the three origin servers will then receive a copy of result from the database, via the transaction mediator, and can return a response to the trusted gateway.

Finally, the trusted gateway, can examine the associated responses to its original, triplicated requests and following another possible round of voting on the web responses, will return the appropriate response to the client.

3.1.4 Required Components

Up until now, a number of components in the KARMA system have been referenced without being clearly defined. In the following section, a detailed description will be given for the three trusted components of the KARMA system and their specific roles in the detection of possible compromises to the system. These trusted components are the gateway, the configuration manager, and the transaction mediator.

Gateway

The gateway is a trusted component that acts as the sole interface between the KARMA system and the outside world. It is responsible for two functions that contribute greatly to the overall security of the system. The first function of the gateway is to conceal its own operating system (OS) fingerprint as well as the fingerprints and real IP addresses of the servers behind it. This forces the attacker to take extra measures to determine the character of the OS, or to mount a blind attack on the system for a variety of OSs. In either case, this additional activity serves to reduce the stealthiness of the attack.

The second function of the gateway, as was shown in the KARMA-1 system, is to provide a trusted proxy that disperses transactions from each client across the set of redundant origin servers. Because the attacker cannot predict what platform, OS, or web server will be used to handle the transaction, he cannot maintain a controlled dialog with any particular platform. In the best case, an adversary would have to successfully land, over time, the proper sequence of transactions on one particular server in order to exploit the system. Once again, this would most likely require suspicious activity that would further reduce the probability of success for such an attack.

Configuration Manager

The configuration manager is a trusted component that never interacts directly with a transaction. Unlike the gateway and the Mediator, it cannot detect compromises in real-time, but rather, detects them over a longer period of time, i.e. dormant attacks set to be triggered at a later time. As was mentioned earlier, the configuration manager plays its most significant role in the first phase of the system, without redundant transactions. Its importance is somewhat diminished in the second phase of development, due to the level of real-time error-checking, however, it is still used and relied upon as a long-term means of detecting a compromise.

The configuration manager's primary responsibility is to periodically check the configuration of the origin servers to detect any attack activity. As a result, an attacker has a limited amount of time to complete an exploit and remove any residual traces of that exploit before the configuration manager detects it. When such an anomaly is detected, the infected server must be taken offline and repaired to a valid configuration state. The configuration assessment on each origin server is performed via a *configuration management agent*, the host-based module, that periodically checks the configuration of its server at different cycles and reports any discrepancies to the configuration manager. The agent is implemented such that it can resist an attack long enough to report this anomaly to the configuration manager.

Transaction Mediator

The transaction mediator provides a trusted proxy for transactions between the origin servers – or more precisely, the set of web scripts running on those servers – and the back-end database. In the KARMA-1 system, the transaction mediator is solely responsible for logging the activity from each of the web servers to the databases. In the case that the configuration manager detects a compromised origin server, these logs can then be used to roll back the database to the last known, healthy state. Logging, however, is not considered necessary when pursuing the KARMA-2 redundant approach because in most cases, voting should preserve the integrity of the database.

The primary function of the Mediator in the KARMA-2 system is to provide a reliable mechanism to vote on the triplicated, dispersed requests going from the origin servers to the database. The most difficult challenge in this is to implement such a voting system without requiring any significant changes to the web servers and scripts. This is important because if the mediator requires the COTS components to be modified significantly, then it would clearly make the KARMA system less portable as it would lead to a complicated and unwieldy installation process.

3.2 Architectural Analysis

In designing the KARMA system, one of the primary motivations was to create an intrusion detection system that would borrow the ideas of masking, redundancy and dispersion from the study of fault-tolerant computing. Each of these factors played an important part in the architectural design of the system and contributed to a final specification upon which a simulation KARMA system was built. In addition to the integration of these concepts from fault-tolerant computing, the design process also involved analysis of the advantages and disadvantages of creating a host-based IDS, network-based IDS or a combination of both.

Given the basic framework of the system, there were a number of different architectural designs that could have been used for this system by varying the set of parameters defined below:

- Voting - The KARMA system utilizes a democratic (i.e. "majority rules") system of voting. As was mentioned earlier, each incoming transaction is first triplicated and then sent to three different origin servers by the gateway. As a result, there are three different places along the transactionary path where voting could take place. These three locations will be referred to as follows:

    1. M_{in} - this refers to voting that is performed by the transaction mediator. It takes place as the transactions (queries) are inbound, or, going from the origin servers to the database.
    2. M_{out} - this stage of voting is conducted by the Transaction Mediator as transactions are headed outbound (i.e. results from a database query).
    3. G_{out} - the final stage of voting is performed by the gateway as the collective responses from the origin servers are being merged and delivered outbound to the external client.

- Redundant Databases - There exist as many copies of the database as origin servers to which a single transaction is delivered. For example in a triplicated network, there would be three redundant databases.

- Dispersion - A sequence of transactions from a given client are sent to different servers or different sets of servers.

- Operating System Variation - The use of different operating systems for each origin server.

3.2.1 Narrowing The Parameters

By varying each of these parameters, it is possible to derive quite a number of possible configurations for the KARMA system. However, some of these can be easily eliminated by making certain educated assumptions about the specific parameters. For the remainder of the design analysis, prospective configurations will be analyzed with respect to KARMA-2 since it is the more fully-featured of the two development phases.

OS Variation

The first parameter that can be eliminated is OS variation. Consider the case of a system in which all of the origin servers shared the exact same operating system and server applications. Although the gateway is equipped with fingerprint masking capabilities, given an intruder with enough time and persistence, there would be a high probability that the intruder would eventually be able to collect a substantial amount of information regarding the configuration of the system. Upon doing so, the intruder could then launch a known-vulnerability attack on the system. Though an intelligent algorithm for dispersing transactions might help to delay the compromise of the system, with enough effort and stealth, an intruder could very possibly gain control of one if not more of the origin servers. Furthermore, the intruder would be able to do this by utilizing only a single attack. Hence, it seems clear that by varying the OSs on the origin servers, the KARMA system would gain a significant advantage in security.

The only scenario in which a system of uniform OSs might have an advantage over a system with varied OSs is a case in which the configuration manager (CM) needed to perform a side-by-side analysis of two origin servers' configurations to detect any anomalous changes. If the system consisted of uniform OSs running identical software, it would be easier to detect any anomalies since nearly everything on each system would have to be exactly the same. However, given that it is possible to develop a CM that could efficiently compare the configurations of two origin servers running different operating systems, it can easily be seen that the uniform OS architecture offers little to no advantage over one with varied OSs. As a result, it will be assumed that the KARMA system always uses origin servers running on a variety of platforms and operating systems.

Voting at G_{out}

The second parameter that can safely be assumed as always true is the gateway voting on outbound responses. It seems reasonable to assume that voting should always be performed at this stage because the gateway is the last possible place where the ill-effects of an intrusion can be caught. The only argument for not employing voting at the gateway would be if it were to have a severely negative impact on the overall performance of the system.

Dispersion

Lastly, an assumption can be made that dispersion will always be used. Based on the "staged attack" assumption that was made earlier, if every transaction is always forwarded to the same machine or set of machines, it would not be long before an intruder, with some knowledge of the system and its parts, could execute a known-vulnerability attack on the system and compromise its origin servers. It is interesting to note though that, in conjunction with the assumption about OS variations, a resulting system without dispersion should still fare relatively well. Given that voting is implemented properly it would appear that even if one of the origin servers were to be successfully compromised by a known attack, the attack would likely have little effect on another origin server running a different OS and software. Therefore, if the ultimate goal of the attack was to corrupt the data going into or out of the system, a democratic voting policy would find the compromised server to yield anomalous results and could thus prevent the attack from committing any further, more permanent damage. Still, this example does not indicate any advantages to using a system without dispersion so it is assumed that dispersion will always be implemented.

Having narrowed down the set of parameters to two possible voting stages and the use of redundant databases, there are just a handful of design configurations left to be examined. Table-3.1, below, illustrates just a few of the possible designs and gives a brief overview of the pros and cons of using each of the respective systems:

3.3 Detailed Design Analysis

Building upon the analysis from Table 1, the following section will present a more detailed analysis of the advantages and disadvantages for each of the remaining design parameters. The section will also discuss the issues that arise as a result of implementing solutions to each of the proposed designs. The remaining design parameters and their subcategories are as follows:

- Voting vs. Non-Voting
  - M_{in}
  - M_{out}
  - G_{out}
- Redundant Databases vs. Single Database

3.3.1 Non-Voting

In the phase one KARMA system, it is not possible to employ voting since there are no redundant transactions taking place. Hence, this system would fall into the class of non-voting. The most noticeable advantage of a non-voting system is that it is simple and does not require the processing time that a voting system would. Depending on

Table 3.1: Brief Architectural Analysis

Red. DB	M_{in}	M_{out}	Pros	Cons
-	-	-	- no mediator needed - fast, no tagging or mediator voting required	- lack of voting makes it harder to detect compromise and corruption
+	+	+	- M_{in} voting makes it easier to detect compromised origin servers - M_{out} voting can detect a corrupted database, and provides secondary defense in addition to G_{out} voting.	- need an effective method to correlate transactions from different sets (tagging) - redundant db adds complexity (requires fault tolerance, synchronization, consistency checking) - slower, because of all of the intermediary voting stages
+	+	-	- faster than voting at all stages, but enough to detect compromises that might not be caught by G_{out}	- without M_{out} can't tell whether the discrepancy comes from a corrupted database or a compromised server
+	-	+	- like the above case, voting at M_{out} can detect those attacks that might not be caught by the gateway	- lack of M_{in} voting allows compromised server to corrupt the database before it gets detected. So even though, it will eventually get caught, it requires the database to be taken offline and rolled back (potentially time-consuming) - also difficult to tell whether discrepancy is from corrupted db or compromised server - potential for a very slow voting algorithm due to the sheer amount of data that could be returned by the database
+	-	-	- faster due to the lack of mediator voting	- without mediator voting, many bad things can happen before they are detected. This will take time to repair.

Red. DB	M_{in}	M_{out}	Pros	Cons
-	+	-	- M_{in} voting will detect attempts to corrupt the database - faster, due to the lack of voting at M_{out}, while not losing any major functionality, assuming correct M_{in} voting	- if database becomes corrupted, requires a full rollback with no backup db - need to implement tagging

the speed of the voting algorithm and the number of voting stages, this advantage is one that could turn out to be significant.

3.3.2 Voting

Voting introduces the ability to compare redundant transactions generated within the system. Because multiple copies of every incoming transaction are randomly dispersed to the origin servers, we can then compare the results of a transaction at various stages along the redundant path (i.e. from the origin servers to the mediator, from the database to the mediator, from the origin server to the gateway) using a "democratic" voting system. The advantage to having a voting enabled system is that it makes it much easier to detect compromised origin servers. Assuming that an intruder's primary intent is to corrupt the database or cause the client to receive corrupted data, it is clear that voting would help to detect and prevent these attacks in almost all circumstances.

3.3.3 Voting at M_{in}

With a single database, voting at the transaction mediator on inbound database requests is required. However, with redundant databases, this vote is optional. The reason for this is that in a single database system, because there is only one copy of the database, if the data were to become corrupted, then a time-consuming rollback process would have to be executed to recover the corrupted data. Furthermore, this would require the entire system to be taken offline while the single database was being restored. In a redundant database system, this would not be as big a problem because each of the triplicated requests could simply be passed to one of the redundant databases. Then, even if one of the databases were to become corrupted, an outbound vote on the database results would most likely catch the error and would be able to rollback the corrupted database. In addition, though this would also require one of the databases to be taken offline, the transaction mediator could then be instructed to divert traffic from the offline database to one of the live ones until it was again made ready to receive further requests from the origin servers.

The most significant advantage to not having voting at M_{in} is that it would greatly simplify the amount of analysis required to secure the system. It would eliminate the two most complex issues of voting at the Mediator, namely syntactic equivalence and tagging. Also, by eliminating these two issues, rather than having to prevent attacks, the KARMA system would be reduced to a system that could survive attacks. However, this in itself is no trivial problem. As a result, in most cases, voting at the Mediator on inbound transactions seems to be a reasonable assumption.

It was previously mentioned that syntactic equivalence and tagging are two of the most complex issues of voting at the Mediator. These two issues are analyzed in greater detail below.

Syntactic Equivalence

The syntactic equivalence problem refers to the issue of voting on requests that are syntactically different but semantically equivalent (i.e. two different requests that may not be word for word identical, but yield the same results). On the mediator, the syntactic equivalence problem deals with voting on the database requests – Structured Query Language (SQL) commands – sent by the origin servers to the database. Given that these queries could have been generated by different scripts running on a number of different web servers on different operating systems, it is quite possible that the queries will be syntactically different but semantically equivalent. For instance, if the scripts were implemented in two different scripting languages by two separate individuals with just a functional specification of what the generated page should display, there would be no guarantees as to the order or syntax of the queries contained in the scripts, even if they were to return identical looking responses. Hence, to minimize the number of false alarms incurred by the voting system, either a highly intelligent voting algorithm must be designed, or additional constraints must be placed on how scripts making database requests are written.

In designing the voting algorithm for the mediator, there are four "special cases" to consider. These are:

- Out of order arguments - two queries that call the same SQL command, but differ only in the order of their arguments
- Out of order queries - a set of queries that produce the same results after being processed by their respective scripts, but are not called in the same order (i.e. Script A - executes queries q1, q2, q3. Script B - executes queries q2, q1, q3. The resulting pages are identical.).
- Variable query counts - One script uses 3 queries to generate a page while another script requires only 2. This can be a result of nested SQL statements (i.e. insert into ratings select user_id, title, movie_id from movies where title='E.T.') vs. $id = select movie_id from movies where name='E.T.' + insert into ratings values ('1', 'E.T.', '$id')).
- Functionally equivalent query sequences - two scripts might have a set of complete different queries that produce the same results. For example a join on tables A and B followed by a join on table C, might have the same effect as a join on tables A and C followed by a join on table B.

Out of Order Arguments Given the number of possible SQL commands that would need to be voted on and the specific number of arguments that each command could take, it is helpful to first examine the full set of SQL commands and determine if there exists a smaller subset that would provide a more reasonable field for analysis.

The first assumption that can be made to narrow down the number of SQL commands is that queries that structurally alter the database or grant permissions to do so must be word for word identical. Included among such queries are those that attempt to create new tables, alter existing tables, or drop them altogether. There are two reasons for applying this constraint to the system. The first is that such commands are "high risk" in that they could very easily be used to corrupt or even destroy all of the information stored in the database. Hence, the likelihood of encountering many of these "high risk" commands for voting is not very high to begin with. Given that any person employing the KARMA system would invariably have security at the top of their list of priorities, there would be a fairly low probability that they would allow web (origin server) administration of their database. It is more likely that the structural administration of the database would be limited to a select few individuals with very high access privileges who would only administer the database from a secure location (i.e. inside the firewall, or from the physical machine itself). However, to maintain the flexibility of the system, rather than forbidding the use of such "high risk" commands altogether, the system could employ the strictest level of voting possible, namely, pure syntactical equivalence. The second reason for applying this constraint to the system is simply that most of the commands that structurally alter the database are highly complex and can take an almost unbounded number of arguments. As a result, an algorithm to compare these queries would most likely be very computationally intensive and would thus degrade the overall performance of the system.

Having assumed that all structurally altering or privileged SQL commands must be syntactically equivalent, the number of SQL commands is greatly reduced to four. The functional description of each of these commands is given below (In the following notation, [] denotes an optional argument, while | denotes a choice, equivalent to OR) [5]:

-
  ```
  SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
      * | expression [ AS output_name ] [, ...]
      [ FROM from_item [, ...] ]
      [ WHERE condition ]
      [ GROUP BY expression [, ...] ]
  ```

```
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
[ ORDER BY expression [ ASC | DESC | USING operator ]
[, ...] ]
[ FOR UPDATE [ OF table_name [, ...] ] ]
[ LIMIT { count | ALL } ]
[ OFFSET start ]
```

- `INSERT INTO table [ ( column [, ...] ) ]`
  `{ DEFAULT VALUES | VALUES ( { expression | DEFAULT } [,`
  `...] ) | SELECT query }`

- `DELETE FROM [ ONLY ] table [ WHERE condition ]`

- `UPDATE [ ONLY ] table SET col = expression [, ...]`
  `[ WHERE condition ]`

From the definitions above, it is obvious that SELECT is the most complicated query of the four remaining SQL commands. It is interesting to note, however, that the SELECT statement makes no changes to the database, but rather can be used only to read from it. As a result, it is possible that for the SELECT statement the system could remain uncompromised even without voting at the mediator since its potential to corrupt the system is low to non-existent. This can be further justified by the following set of arguments. First, it must be noted that in most cases only a single database query would be sent to the database from the initial three. This is because the system would not want to execute an insertion into the database three times as a result of the redundant origin server requests. However, the SELECT statement introduces a unique alternative to all other SQL commands based on the initial observation that it makes no changes to the database. Namely, all three of the redundant queries can be sent to the database without causing any significant threat to the performance of the system. Using this new information, given two SELECT statements that are syntactically different, one of two possible scenarios could occur without voting at the mediator. Either the statements were crafted by two different individuals (on different origin servers) but yield the same response pages – should pass voting at the gateway with no problems – or one of the statements was altered by an intruder resulting in a corrupted response page that could easily be detected by voting at the gateway. Clearly, neither of the scenarios appear to be very harmful, as neither one would result in any permanent damage to the system nor return falsified information to the user.

Still, it would be helpful to investigate what an equivalence algorithm for the SELECT command might entail. The first thing to notice is that a number of the lowercase expressions (variables) are repeated several times throughout the entire SELECT command. So the algorithm would benefit greatly, if functions could be written for each distinct variable type. These functions could then be reused for every expected occurrence of a specific variable. In other words, an equivalence function for the variable type *expression* might be defined as follows (pseudocode):

```
bool expression::compare (expression expr) {
   each expression object (THIS, expr) would be in the form: a, b, c$\ldots$
   parse THIS -- the current expression object -- and expr into an
   array of strings by pulling out the comma-delimited values
   verify that the size of the arrays for THIS and expr are identical
   and that each of the array's values is unique
   for each string in the array for THIS
      if the array for expr contains that string
         loop
      otherwise, return false
   end for
   return true
}
```

By defining such a function, comparing equivalence for a SELECT statement would be simplified to looking for the keywords, ALL, DISTINCT, FROM,...,FOR

UPDATE OF, and storing each of the values following these keywords into their own variable type. The equivalence method for this SQL object would then consist of an iterative execution of its member variables' equivalence functions. For example, given the equivalence function for the variable type *expression* above, the equivalence algorithm for SELECT might appear as follows:

```
bool select_obj::compare(select_obj sel) {
   from the definition of SELECT, we know that the "main statement,"
   GROUP BY and ORDER BY will all be of the variable type, expression.
   ...
   main_expr->compare(sel->main_expr);
   ...
   group_by_expr->compare(sel->group_by_expr);
   ...
   order_by_expr->compare(sel->order_by_expr);
   ...
}
```

What this essentially points to is a very modular structure for each SQL command. From this conclusion, the next step in designing an equivalence algorithm would be to model the SQL command as an object containing members of various SQL variable types. If this could be done effectively (i.e. modeling a SQL command and its sub-expressions as classes in C++), then testing equivalence for a SQL command would be as simple as parsing it into the object and then comparing the values of its member variables.

To further investigate the possibility of creating these SQL objects, it might be beneficial to analyze each variable and its possible values to determine whether mapping it to standard C types would prove to be a feasible task. An analysis of each variable type, its possible values and their "mappability" is described in Table-3.2.

Given this mappable representation, designing a SQL object would then be straightforward and an equivalence test could be programmed with very little difficulty. This

Table 3.2: SQL Variable Types

Variable Type	Possible Values	Mappability
expression	A table's column name or expression	Easily mapped as a char *. However, since most expressions would appear in a list, this might be better represented as an array of strings, or char **.
output_name	Specified another name for an output column	This could be mapped as a string (char *) but it would most likely be stored along with its associated expression, i.e. "movie_title as title" could be stored as a single expression.
from_item	A table, sub-SELECT, or JOIN statement	Each table name, sub-SELECT statement, JOIN statement could be stored as a string, resulting in an array of strings (char **).
condition	A boolean expression in the form of "expr cond expr," or "log_op expr."	Mappable as an array of strings, consisting of expressions and conditionals, however, to determine the equivalence of two conditional clauses might prove to be quite difficult later on, in that a great deal of calculation and knowledge of logical equivalence would be required (i.e. not (p and q) = not p or not q). Hence, a constraint might need to be set that where clauses need to be strictly equivalent and they could then be stored as a single string (char *).
select	Another select statement	This could be easily mapped given that a valid representation for a select object had been defined
operator	An ordering operator (i.e. <, >)	This could be mapped as a string (char *).
table_name	The name of an existing table or view	Mappable as a string (char *).

Variable Type	Possible Values	Mappability
count	Maximum number of rows to return	Can be mapped as an integer (int).
start	Number of rows to skip before returning values	Mappable as an integer (int).

can best be illustrated by the following example. Since the SELECT statement is quite complex, the simpler DELETE statement will be used. The DELETE command could be modeled as an object in the following manner (C++ pseudo code):

```
struct delete_obj {
   bool only;
   char *tablename;
   char *where;
};
```

A corresponding equivalence test would then be implemented as follows:

```
bool test_equivalence(delete_obj *do1, delete_obj *do2) {
   if (do1->only != do2->only) {
      return false;
   } else if (strcmp(do1->tablename, do2->tablename) != 0) {
      return false;
   } else if (strcmp(do1->where, do2->where) != 0) {
      return false;
   }
   return true;
}
```

Not every equivalence test would be this straightforward, nor would every SQL command be best represented using the mappings given in Table-3.2. For example, the expressions and values in an INSERT command would probably be better represented as a set of (key, value) pairs in a hash table, rather than as arrays of strings. This would enable the equivalence algorithm to then simply look up keys and verify their values rather than trying to determine a one-to-one mapping between the array indices – since the keys themselves might appear out of order in the two original queries. Recognizing though, that there might be differences in the representation of individual objects, it is relatively clear that, by successfully modeling a SQL query as an object, the problem of out of order objects could be solved with relative ease.

Out of Order Queries Out of order queries deals with the case in which one developer might call a set of queries in the order A-B-C, while another developer might call them in the order B-A-C. One algorithm that might work to effectively solve this voting problem would be to cache the database requests until the two scripts had sent all of their queries to the mediator. Then, once all of the queries had been submitted, the mediator could perform a matching between two origin servers' requests to determine if they had called the same set of queries, but in a different order. There are a number of problems with this solution though. First of all, there would be no means for the mediator to determine when a script was finished sending all of its queries to the database. Secondly, it might be the case that a script, making a set of database queries, would need to use the data retrieved for one query before it could go on to making the next request. As a result, if the system were to wait to cache up all of the requests for a particular script before voting, it would effectively go into a deadlock situation because each of the scripts would wind up waiting for its first query to execute, while the mediator would be waiting for the script to move on to its next query. Hence, with the exception of the mediator actually looking up the script's source on the origin servers, or maintaining a list of exception pages, it would not be possible for voting to occur over a range of queries rather than just on a one-to-one basis. Without a hefty set of constraints, for example, requiring that all scripts make all database requests at the beginning of a script, with no cross-dependencies, there is no clear solution to the problem of implementing an equivalence algorithm for out of order queries. Rather, it would seem the system would need to be constrained in so far as requiring queries to be performed in the same order across all origin servers.

Variable Query Counts For the same reasons as out of order queries, there is no feasible solution to comparing a set of queries across two different origin servers in real-time, especially given that the number of queries differs.

Functionally Equivalent Query Sequences Finally, the problem of functionally equivalent query sequences also falls into the category of voting across origin servers on a range of database requests. In real-time, there would be no means of acquiring all of the requests before initiating a voting sequence without either stalling an entire set of scripts, or forcing the queries to be independent of one another and then to be called in sequence. Even if such a constraint were placed on the system, by virtue of the sequential nature of most scripting languages, the script would not be able to proceed to the next query without receiving a response back from the database (in our case, the mediator). Hence, such a constraint would most likely require further modification of the scripting language's source code. It seems that a better solution would be to simply outline the set of SQL calls that would need to be made for a particular script and to make them uniform in order and in purpose across all of the origin servers.

Fully Constrained SQL Queries Up until now, consideration has only been given to scripts containing unconstrained or partially constrained SQL queries. However, one might wonder if implementing such an equivalence algorithm for voting on SQL queries wouldn't just add more complexity to the system. It would seem that using a fully constrained set of queries might provide the same security benefits without the extra performance cost of creating SQL objects and determining their equivalence in that manner. Thus far, the most important reason why emphasis has been placed on limiting the number of constraints in the system is that diversity often enhances security. This is evidenced by the use of dispersion, varied OSs, and varied web server software. However, the security gained in diversity with respect to queries would mostly likely come from changes to the scripting language used, or to the scripts themselves. The wording of individual SQL queries within a script would have no real detrimental effect on the level of security, but the constraint might simply come as a nuisance to an independent developer writing the scripts for one of the origin servers. However, to another developer, the same constraint might be seen as a blessing in that, SQL queries can often grow to be quite complex, as was illustrated by the description of the SELECT statement above. Hence, it seems that developing a system with fully constrained SQL queries might also be a viable choice as a solution to the syntactic equivalence problem.

In summary, the syntactic equivalence problem is very difficult to solve completely. However, there are a number of possible solutions and steps that can be taken to allow voting to occur with a minimal number of false alarms, while not requiring all scripts to be fully identical.

Tagging

Tagging is a function that is mandatory if voting is to be used at the mediator on inbound database requests. Upon receiving an incoming transaction, the gateway triplicates each request and then sends it to three different origin servers. Each of these origin servers could then call up a script that might make a number of requests to the database. The question that arises is how to correlate those requests at the mediator when hundreds of requests might be coming in per second from any number of different origin servers. This calls for a method of transparently labeling each database request such that the mediator will be able to identify it as belonging to a unique set of requests. The database could then wait for its triplicated counterparts to arrive before voting and, if successful, forward the final request to the database.

A proposed solution to this problem, utilizes the fact that the CGI interface, by definition, takes all HTTP header lines from the client and then places them into the environment with the prefix HTTP_ [3]. Hence, by adding a custom tag into the list of header lines at the gateway, it would be possible to get a tag through to the scripts without having to modify any of the software involved. Furthermore, given that the scripts could spawn off a database query without starting up a new process, a custom-made database driver manager installed on each origin server would then be able to pull the tag from the environment without the script ever knowing. Once the tag had been pulled from the environment the driver manager could then send this tag along with the original query to the transaction mediator, which would then use the tag to uniquely identify this request along with its triplicated counterparts.

Having been presented with the arguments both for and against voting at the mediator, it seems reasonable to say that voting on inbound transactions at the mediator would provide greater security, while also not being so complex as to slow down the entire system. In addition, though it was previously stated that voting would be optional for a system with redundant databases, the increased gain in security would most likely provide a better option than freely allowing the database to become corrupted, requiring a timely and costly rollback procedure to be implemented as well.

3.3.4 Voting at M_{out}

Voting at the mediator outbound is probably the most flexible of all the stages of voting in that it has the least impact on the actual security of the system. Furthermore, it is only required if a redundant database is used. Assuming that voting at the gateway outbound and the mediator inbound are in working order, it is clear that incorrect transactions will never be sent out to the client or written to the database - except for the case in which an intruder manages to plant three dormant attackers on three separate origin servers that are programmed to fire up only when they are simultaneously selected, allowing them to falsely pass all voting stages undetected. As a result, voting at the mediator outbound proves to be solely a matter of convenience. It provides an advantage in that it makes it easier to determine, during a failed vote, whether the corruption was a result of the database, or the origin servers. Furthermore, if a single script requires multiple transactions with the database, or makes various calls to other scripts, it may help to determine which portion of the script, or script files was corrupted. In conclusion, voting at the mediator on responses being sent back to the origin servers seems to be unnecessary. Furthermore, it is a luxury that would most likely turn out to be more costly than it is worth as the size of the responses being sent back to the origin server can vary greatly, limited only by the size of the database. If a simple bit-for-bit comparison was performed on two different database responses to determine equivalence, it is not difficult to imagine the negative effects that this outbound voting could potentially have on the performance of the entire system.

3.3.5 Voting at G_{out}

Earlier an assumption was made that voting would always be used at the gateway on outbound transactions. If voting at the gateway were not always used, it would be possible for an intruder to compromise a server in such a way that only transactions going out to the client would become corrupted. Then, the compromise would not be detected by the mediator going inbound or outbound, and without voting at the gateway, would be free to send out corrupted data completely undetected.

Like voting at the mediator on inbound transactions, voting on outbound transactions also gives rise to a number of issues. The most significant of these issues is again, syntactic equivalence.

Syntactic Equivalence

The syntactic equivalence problem for outbound transactions at the gateway (essentially HTTP responses) is very similar to the syntactic equivalence problem for database requests in that an algorithm must be determined to effectively vote on the responses from three different origin servers that may vary slightly as a result of architectural, web server, database, or programmer differences. Listed below, are some of the most common discrepancies that might arise as a result of these differences:

1. Whitespace/Carriage Return - it is common for different architectures to use different characters to denote a new line. Windows machines may use just a '\n' character while it is more common for UNIX systems to use '\r\n'. In addition, the whitespace problem refers to the fact that different script writers might choose to space their documents differently, this could be the result of tabbing, extra spaces, or even extra carriage returns.

2. Character Set - "Fundamentally, computers just deal with numbers. They store letters and other characters by assigning a number for each one." Over time, hundreds of different encoding systems to map numbers to characters. In addition, no single encoding has been large enough to capture the entire range of possible characters. As a result, single languages alone have often required multiple encoding systems. Unfortunately, the sheer number of encoding systems in use today has inevitably led to conflicts between the different encodings – using the same number for two different characters, or vice versa. Furthermore, this has created the need for servers to support many different encodings, and oftentimes being able to map characters from one encoding to another. Today, there is a new standard being developed called Unicode. "Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others." [18]

3. Numerical Rounding - It is possible that two different scripting languages, or in the case of redundant database, two different database servers might treat numbers differently. For example, an MySQL database might store the value of ".999..." as "1.0," whereas an Oracle database might store it as the actual number ".999..."

4. Case-Sensitivity - This would occur simply if two different programmers had written a set of scripts using different case formatting for their generated HTML pages. One developer might have used all upper-case tags, while another might have preferred to use lower-case ones.

Whitespace/Carriage Return An equivalence algorithm could be designed to solve this problem simply by ignoring repeated whitespaces and carriage returns. Instead, a set of connected whitespaces and carriage returns could be defined as a chunk, thus yielding, characters in a document separated by chunks rather than individual whitespaces and carriage returns. So "The   Draper Laboratory KARMA System(carriage return)," would be treated as "The(chunk)Draper(chunk)Laboratory(chunk)KARMA(chunk)System.(chunk)" However, it is important to note that some of these whitespaces might occur inside string values, i.e. (""), where they should not be ignored. For example, a page with a submit button with a value of "Push Here" should not be equivalent to a page with a submit button labeled "PushH ere." By utilizing this fairly simple algorithm, most cases of the whitespace/carriage return problem would be effectively handled by the KARMA system.

Character Set Due to the sheer magnitude of character encoding systems that are currently in use, it seems unlikely that an algorithm could be designed to handle variations in character sets produced by the web servers. However, for the purposes of the KARMA system, we will consider a single encoding, namely the ISO-8859-1 (Western European) character set. As of now, ISO-8859-1 has been the most commonly used encoding system for the purposes of the web [19]. The reason for this is that, it is supported directly by, or is at least fully compatible with, nearly all flavors of UNIX and Windows. In addition, it is mostly compatible with the Macintosh, though a few incompatibilities (fourteen known problematic characters) have been discovered over time [2]. Due to the fact that the incompatibilities with Macintosh (Mac) are well-known and identifiable, one possible solution to handle the equivalence of different native character sets would be to ignore those characters that are known to be incompatible with the ISO-8859-1 standard. Alternatively, a mapping of characters could be defined between the Mac's native storage code and the ISO-8859-1 standard. This could then be used in the equivalence algorithm to compare the mapped values for Mac's native characters to the ISO-8859-1 standard ones. The final possible solution would be simply to apply a constraint that the set of conflicting characters (fourteen known), would not be handled by the KARMA voting system.

It may seem like a huge shortcoming to only have considered the single ISO-8859-1 encoding, but with the recent adoption of the Unicode standard on many different platforms, it seems likely that the character set problem will eventually become a moot issue.

Numerical Rounding The numerical rounding equivalence problem is not as difficult as it would seem. An efficient algorithm could be designed to simply regard two different numbers as equivalent, if they were close enough to one another, where closeness would be determined by a predefined value. Such an algorithm will be further described in the implementation chapter to follow.

Case-Sensitivity Finally, the case-sensitivity problem can be dealt with by simply allowing the equivalence algorithm to compare all alphabetic characters with both cases. For example, "abCd" would be equivalent to "AbcD." From an implementation standpoint, this could be done simply by setting every alphabetic character to its lower case counterpart prior to performing the equivalence test.

Clearly, the above examples are just a few of the issues that might arise in the voting on outbound HTML responses at the gateway. However, it seems that an equivalence algorithm could be effectively designed (described in Chapter 4) to catch most of the differences that might ordinarily arise, thus minimizing the number of false alarms while also providing the KARMA system with a secure and effective voting mechanism.

Voting Hashes

A final issue to be considered in the case of outbound voting at the gateway, is a possible improvement in performance that could be gained by voting on hashes of the pages rather than the actual pages themselves. Clearly, this mechanism would only be useful for voting on static pages, since dynamic pages would be subject to change upon every subsequent request.

The HTTP 1.1 specification describes an entity-header field of Content-MD5 that is included in the HTTP response headers, depending on how the web server is configured, and contains the value of an MD5 hash of the generated page. If we were to cache the values of the MD5 hashes for all static pages (generated upon installation and updated as changes are made), then upon receiving a request at the gateway for one of those static pages, rather than triplicating the request, it would be possible to make just a single request. Then, when a valid response was returned by the origin server, the gateway could simply extract the MD5 hash value from the response header, and then compare this with the cached hash values.

The gain in performance would come mostly from not having to triplicate the web requests, and also from being able to run the equivalence check on a single 128-bit string, rather than having to do a character-by-character comparison on a response page of uncapacitated length.

Clearly, it seems as though voting at the gateway would greatly enhance the security of the KARMA system and its ability to detect intrusions. In addition, the solutions to many of the issues with gateway voting seem to be solvable within a reasonable amount of time. Hence, the final KARMA system should definitely include a voting stage on outbound transactions at the gateway.

3.3.6 Redundant Databases

Assuming that voting at the mediator inbound is present and working properly, it seems there is no need for a redundant database since the transactions going into all three databases should have identical results. However, it might still be worth it to use redundant databases for the gains in fault tolerance in case of a database failure. Additionally, in the event that one of the database servers does fail, having the redundant databases would provide an alternate means of re-routing the database requests without having to take down the entire system.

In the case that voting at the mediator inbound is not present, the redundant database proves to be very useful in conjunction with the outbound voting at either the gateway or the mediator. While the lack of an inbound voter allows the database to become corrupted, the redundant database/outbound voter combination effectively allows the system to detect and repair this corruption (via an expensive rollback procedure) before it becomes visible to the client.

For the above stated reasons, it would appear that a redundant database would always be the more desirable choice. However, this is not necessarily true. There is one significant downside to having redundant databases: consistency. In order to maintain three separate databases, the system must effectively be able to ensure that the three databases are consistent at all times. For example if a triplicated set of requests were to be sent to the transaction mediator, a vote might take place, and then the queries would be sent on to the database servers. If the queries consisted of an INSERT statement, then clearly, they should leave the database having added a row to one of the existing tables. However, if one of the queries was to fail, it would send a response back to the mediator which would then pass it on (no voting at M_{out}) to its respective origin server. Similarly, the other two origin servers would receive their correct responses. Upon voting at the gateway, one of two things might happen: 1. The database call in the script would have returned an error message, in which case the page would be voted differently and an intrusion error would be thrown. 2. No error will have been thrown, and the page will pass voting successfully (INSERT does not return any visible data). In the first case, the KARMA system would then need to determine where the error occurred (within the script, or within the database) in order to restore the consistency of the database corresponding to the failed request. In the second case, the system would most likely detect the error at some later point, at which time it would need to perform the expensive rollback procedure to restore all of the databases to a consistent state.

The above example, is just one case in which the consistency of the three database servers could be damaged. There are many other ways in which problems could occur in maintaining the three redundant databases. As a result, in spite of the gained fault-tolerant capabilities, the system would also grow in its complexity and would also lose some performance in the long run (since the system is not doing any workload distribution amongst the redundant databases there is no real performance gain).

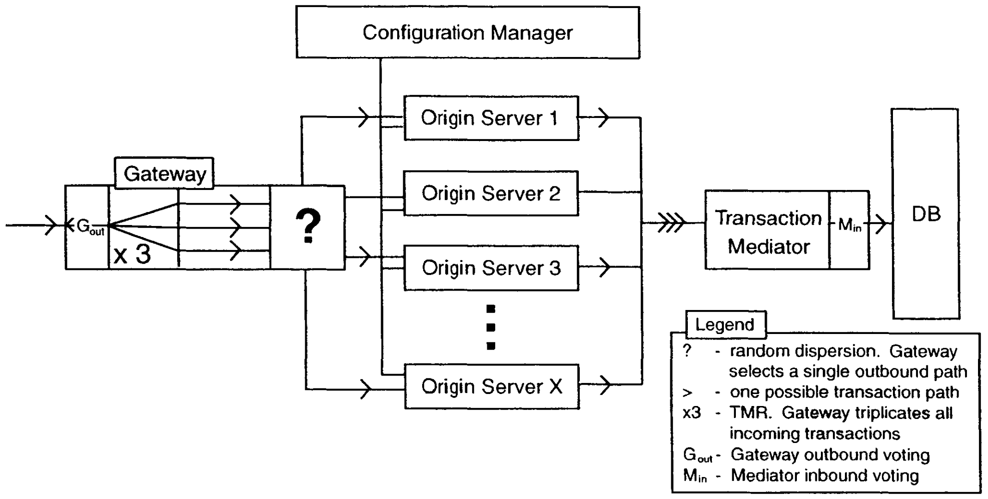
3.3.7 Single Database

Given that voting at the mediator will always be used on inbound database requests, it seems that a single database system would provide a very workable and secure solution. While not providing the conveniences of on-the-fly re-routing or backup capabilities, as a redundant database would, the single database system still allows for a relatively reliable and much simpler implementation, without the complexities of maintaining the consistency and integrity of a redundant system.

3.4 Design Conclusions

The final implementation of the KARMA system is shown in Figure-3-3.

3.4.1 Final KARMA System



Configuration Manager
Origin Server 1
Origin Server 2
Origin Server 3
Origin Server X
Gateway
G_{out}
x 3
?
Transaction Mediator
M_{in}
DB
Legend
? - random dispersion. Gateway selects a single outbound path
> - one possible transaction path
x3 - TMR. Gateway triplicates all incoming transactions
G_{out} - Gateway outbound voting
M_{in} - Mediator inbound voting

Figure 3-3: Final KARMA System

As shown in Figure-3-3, the final KARMA system consists of a gateway, dispersing transactions using the triplicated module redundancy algorithm to a set of origin servers. These origin servers are monitored by a configuration manager with the help of agents running locally on each machine. The origin servers also have the ability to send requests to a single database via the transaction mediator. Voting occurs at the gateway on outbound transactions and also at the mediator on inbound database requests.

As a proof-of-concept, both the KARMA-1 and KARMA-2 systems were partially developed. The first phase was developed by a group working at the Charles Stark

Draper Laboratory. In addition to implementing a near full version of the KARMA-1 system, a simulation of the KARMA-2 system was also developed for the purposes of testing and to show that such a system was indeed realizable and implementable. The details of each implementation will be described in the following chapter.

Chapter 4

Implementation

The implementation was conducted in two phases. The purpose of the phases was not so much to serve as a progression, but to test out different pieces of the system, and to demonstrate a proof-of-concept for the KARMA system.

4.1 KARMA-1

The initial phase of the KARMA system was implemented without using redundancy or voting. The system did use dispersion, however, to enhance its overall security. In addition, a network-based configuration manager was implemented along with a host-based agent that was installed on each of the origin servers. The KARMA-1 system consisted of a single gateway, followed by four origin servers – all being monitored by the configuration manager – followed by a single database machine.

The specifications for each of the machines used were as follows:

The primary objective of the KARMA-1 system was to test the advantages of dispersion and OS variation on the origin servers. In addition, the configuration manager was tested for its performance abilities.

Table 4.1: Phase One KARMA Configuration

Component	Architecture	OS	Server Software
Gateway	Intel Pentium III	OpenBSD	N/A
Configuration Manager	Intel Pentium III	OpenBSD	N/A
Origin Server 1	Intel Pentium III	Redhat Linux 7.2	Apache v1.3.24, PHP v4.1.2
Origin Server 2	Intel Pentium III	Windows 2000	Microsoft IIS5, PHP v4.1.2
Origin Server 3	Sun Ultra 1	Solaris 8.0	Apache v1.3.24, PHP v4.1.2
Origin Server 4	Intel Pentium III	Windows NT Server	Microsoft IIS4, PHP v4.1.2
Database	Intel Pentium III	Redhat Linux 7.2	IBM DB2

4.1.1 Gateway

Dispersion was implemented at the gateway such that all of the headers from an incoming web request would be scrubbed out before the request was dispersed randomly to an origin server. The scrubbing mechanism would effectively eliminate all headers except for those necessary to make the request. The scrubbing of headers played two functional roles in the system. For incoming transactions it was required so that extra information could not be put into the headers that might either trigger an attack, or contribute to one. On outgoing transactions, the scrubbing of headers enabled the system to remain fully anonymous (fingerprint-masking) by replacing all of the specific identification headers with generic non-descriptive values.

In addition to the elimination of potentially hazardous headers, the dispersion mechanism was also responsible for distributing transactions such that the risk of a "staged attack" might be minimized. This was done by using a pseudo-random dispersion algorithm. Essentially this meant that transactions would be distributed randomly to the origin servers, but, the system would further ensure that no two consecutive transactions, from a given client, could ever be sent to the same origin server.

4.1.2 Configuration Manager

The final step in the implementation of the KARMA-1 system was to develop a configuration manager that could effectively monitor the activity on each origin server and trigger a "reset" if any anomalous behavior was detected. Rather than monitoring every type of action performed on a machine, the responsibility of the CM agents was limited to polling the web server data and scripts to detect any modifications to those files. Assuming that those pages and scripts were originally designed to generate identical web responses, it was observed that by periodically verifying their integrity it might be possible to prevent any significant corruption to outbound responses or inbound database requests.

4.2 KARMA-2

The KARMA-2 system differed from the KARMA-1 system in that it integrated redundancy into the system. In addition to dispersing packets pseudo-randomly to the origin servers, the KARMA-2 gateway also employed TMR (triple modular redundancy) and would triplicate each incoming transaction before dispersing it to the origin servers. Each of the recipient servers would then independently process the requests and, if database access was required, would send another request to the mediator or simply return a response to the client through the same redundant path. The advantage to using a redundant system was that voting could now be employed.

As was mentioned earlier, the KARMA-2 system was not designed to serve as a fully-fledged version of KARMA-1, but rather was implemented to test the advantages of using redundancy, independent of any other factors (OS variation, configuration manager). Therefore, the entire KARMA-2 system was implemented running on a single machine running a gateway, three origin servers, a transaction mediator and a database. The specifications of each of the components used were as follows:

Table 4.2: Phase Two KARMA Configuration

Component	Architecture	OS	Server Software
Gateway	Intel Pentium III	Redhat Linux 7.2	N/A
Origin Server 1	Intel Pentium III	Redhat Linux 7.2	AOLServer v1.3.24, PHP v4.1.2
Origin Server 2	Intel Pentium III	Redhat Linux 7.2	Apache v1.3.24, PHP v4.1.2
Origin Server 3	Intel Pentium III	Redhat Linux 7.2	Apache v1.3.24, PHP v4.1.2
Transaction Mediator	Intel Pentium III	Redhat Linux 7.2	N/A
Database	Intel Pentium III	Redhat Linux 7.2	MySQL

4.2.1 Libasync - An Asynchronous Socket Library

Libasync is a library included in the self-certifying file system (SFS), a DARPA sponsored project that implements a secure, global network file system with completely decentralized control. The primary focus of the libasync package is to simplify the use of asynchronous sockets over a network using TCP. It also takes advantage of non-blocking input/output in order to improve the performance of reads and writes over sockets [10].

The libasync package was chosen to implement the gateway and the transaction mediator because of its functionality and ease of use. By using the libasync package, the writing of an asynchronous gateway and mediator was greatly simplified, as the asynchronous socket library provided utilities for "creating sockets and connections and also for buffering data under conditions where the *write* system call [could] not write the full amount requested." Additionally, one of the most useful features of the libasync package was its ability to easily register callbacks that could be used to trigger events on a socket in association with a specified condition of readability or writability. The callback features and non-blocking I/O made it much easier to implement a gateway that could simultaneously read multiple requests from a client while also writing a set of triplicated requests to the various origin servers. In the case of the mediator, it also simplified the task of reading in multiple database requests while writing back responses at the same time.

4.2.2 Gateway

The gateway was implemented using the libasync package. The flow of events is best described by an outline of the code that was used. First, a description of the objects that were defined and used will be given. There were three primary classes that needed to be defined to implement the gateway. These were the vote_entry, connection, and gateway classes:

- vote_entry - This class was a linked-list structure used to maintain a cache for the votes (responses) that had been received for a given ID thus far. There was a single voting cache used for the entire gateway system consisting of a linked-list of vote_entry objects. The member variables and functions for the vote_entry class were as follows:
  - fileid - the ID associated with the current vote_entry
  - respcount - the number of responses that had been received thus far
  - next - a pointer to the next vote_entry
  - prev - a pointer to the previous vote_entry
  - insert() - a method to insert new entries into the voting cache
  - remove() - a method to remove an existing entry from the voting cache
  - find() - a method to find an existing entry based on a given ID
- connection - The connection object was responsible for maintaining all of the information relevant to a single triplicated request. For example, an incoming transaction received by the gateway would in turn create three different connection objects. The member variables and functions for the connection class were as follows:
  - host - used to store the host for the current connection

- port - used to store the port for the current connection (80 on most normal webservers, set to 8000, 8001, and 8002 on the simulation KARMA-2 system for the 1st, 2nd and 3rd origin servers respectively)
- request_id - the ID/TAG associated with the current connection
- vote_id - an id taking the value of 1, 2, or 3. This value was simply used to distinguish this connection with respect to the other two connections in the triplicated set.
- htreq - a copy of the HTTP request object
- htresp - a copy of the HTTP response object
- deof - a boolean flag to signal whether an end of file has been read from the origin server
- delay_req - a timeout for reading from the gateway
- delay_rresp - a timeout for reading from the origin server
- dsock - the socket handle for reading and writing to the origin server
- fcreated - a boolean flag to signal if the temporary response file has been created
- gotresp - boolean set to true if the headers had been successfully parsed out of the HTTP response from the origin server
- resp_buf - a temporary storage buffer used to store the response while it was being partially written to the file system
- connection() - method to create a new connection object
- connection() - method to destroy the current connection and all of its registered callbacks/timers
- get_filename() - method that returned the filename to which the temporary HTTP response for this connection was written

• webproxy - The final object was the webproxy (gateway) object. Upon receiving an incoming transaction, the gateway would create a new webproxy object. The webproxy was then responsible for managing a set of connection objects that were created once a valid request had been received. This request would then be triplicated and sent to each of the connection objects. The member variables and functions of the webproxy object were as follows:

- connections - an array used to store the three connections
- htreq - a local copy of the HTTP request
- htresp - a copy of the HTTP response to be sent back to the client
- gotreq - boolean set to true if a valid request header was received
- gotresp - boolean set to true if a valid response header was received
- conn_id - long integer used to store the TAG associated with this transaction. This value was used to support the implementation of tagging.
- delay_vote - a timeout used to ensure that action would be taken if all three response were not received in a reasonable amount of time
- delay_wresp - a timeout for writing back to the client
- osock - the socket handle for reading and writing to the client
- req_buf, resp_buf - temporary buffers used to store the request and response while they were being written to the origin servers and gateway, respectively.
- oeof, veof - boolean variables used to signal that an end of file had been read from the gateway, or that voting had been completed.
- webproxy() - used to create a new webproxy object
- webproxy() - method to destroy the existing webproxy object along with all of its registered callbacks and timers
- timeout_vote - the timeout method for voting
- timeout_wresp() - the timeout method for writing a response to the client
- timeout_req() - the timeout method for reading a request from the gateway

- timeout_rresp() - the timeout method for reading the response from the gateway
- err_handle() - a method for handling errors and returning an error web response if necessary, or destroying the object along with its registered callbacks/timers
- connected() - a callback method that was called once the gateway had successfully established a connection with the origin server
- duplicate_request() - a method that was used to clone the HTTP request object and then send it to the respective connection objects
- make_requests() - a method that was used to create connections to the three origin servers and then send out a copy of the original HTTP request to each one.
- read_resp() - a callback method that was triggered if there was information waiting to be read from one of the origin servers. The webproxy was then responsible for reading the response into a temporary buffer.
- read_req() - a callback method that was notified if there was information waiting to be read from the client. The webproxy would then read an HTTP request from the client.
- write_resp() - a callback method that was triggered if socket to the temporary file was writable. The webproxy would then write all of its buffered response data to a temporary file.
- write_req() - a callback method that was started if the socket to a given origin server was writable. The data in the request buffer would then be written to the origin server.
- write_error() - a method that was used to generate a fake HTTP response page useful for displaying an error message to the user
- write_final_resp() - a callback method that was notified if the socket to the client became writable. It would then write an appropriate HTTP

response to the client socket.

- write_error_resp() - a callback method that was triggered if the socket to the client became writable. It would then write the generated error message to the client socket.
- vote() - a method that was called once all three connections had returned a valid response and written it to their temporary files. The voting algorithm would then be run on the three saved responses to determine their equivalence.

Given this description of the primary objects and their methods, the gateway can best be modeled by using a block diagram. This diagram is shown in Figure-4-1.

Essentially, the gateway, upon receiving a request would generate a custom tag (in this implementation, simply a counter). The gateway would then create a new webproxy object and insert an entry into the voting cache using the custom tag value. Once the entire HTTP request had been read, the gateway would then attempt to establish a connection to the three origin servers. If all three connections were successfully established, the gateway would then duplicate the original HTTP request and would pass it on to the three origin servers. Upon getting back the first response from the triplicated set of requests, the gateway, rather than writing it back to the client immediately, would write the entire response to a temporary file. It would then look up the tag value for the received response and would increment its counter in the voting cache. The system would then wait for the remaining responses. After getting the third and last response, the counter would be incremented again yielding a final count value of three. This would signal to the gateway that all of the responses had been received and it would then proceed to vote. The voting algorithm basically consisted of reading through a set of temporary response files and comparing them character-for-character while also accounting for a subset of the issues described in the previous chapter, namely whitespace/carriage return, number-rounding and case-sensitivity issues.

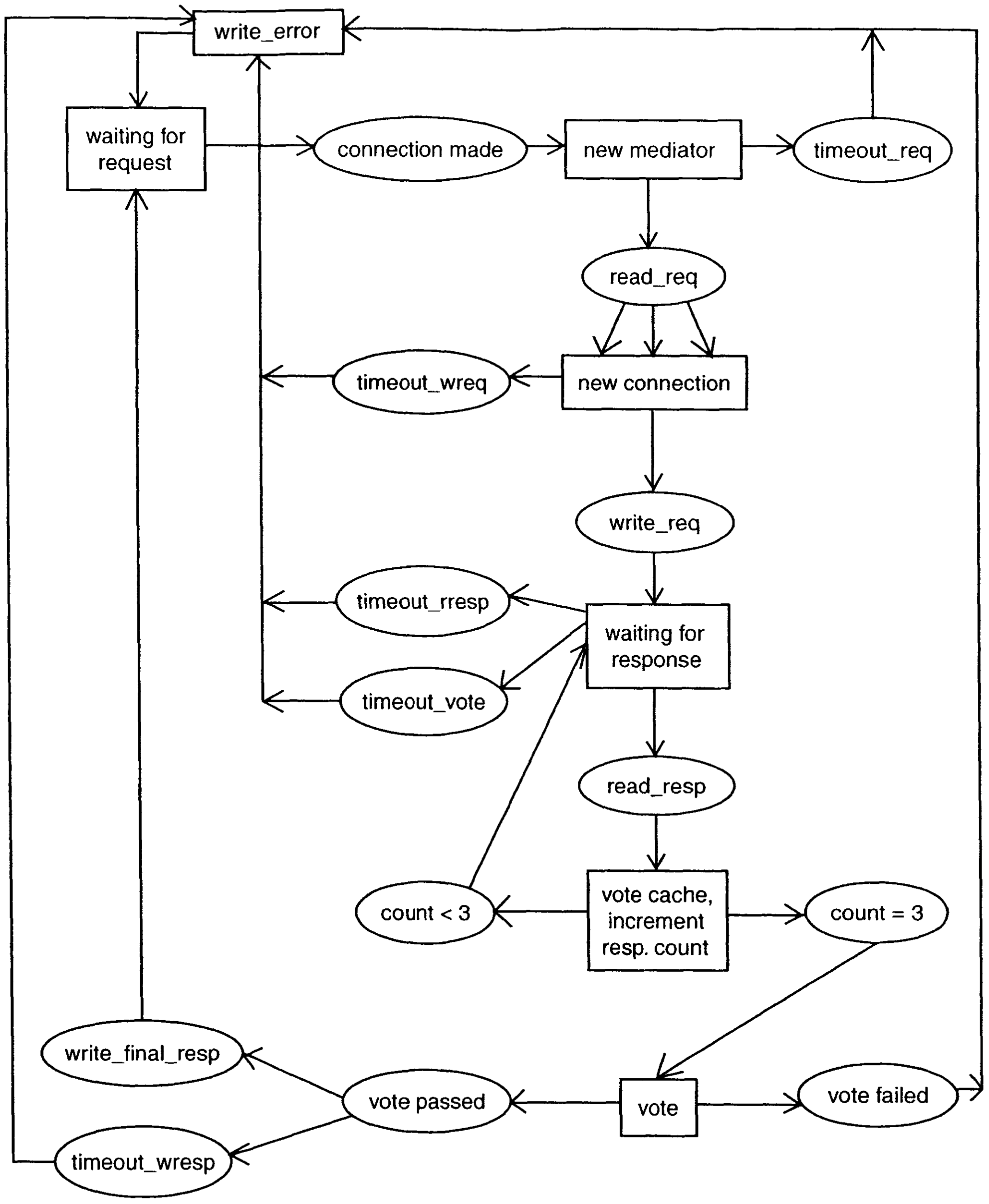


Figure 4-1: Gateway Block Diagram

Syntactic Equivalence

The implementation of the KARMA-2 system dealt with the problem of syntactic equivalence at the gateway much as it was described in the design process. In its simulation state, the equivalence algorithm was able to effectively handle the numerical rounding issue, the case-sensitivity issue, and the whitespace/carriage return issues.

Whitespace/Carriage Return The whitespace/carriage return issue was handled exactly as it was described in the design section. As the two response files were compared character-for-character, the system would raise a flag upon encountering a whitespace or carriage return (included characters were '\t,' '\r,' '\n,' ' '). All successive whitespaces or carriage returns would then be ignored until a legitimate character was retrieved. Voting would then be resumed as usual. In addition, if a quote character (single or double) was encountered, then the system would not raise the flag to ignore whitespaces. Only after the second quote character (closing quote) was encountered, would the whitespace ignoring functionality be re-enabled.

Numerical Rounding The numerical rounding problem was handled as follows:

1. While looping through character-by-character between the two temporary response files
2. If a numerical character is encountered in both files (i.e. '-', '0-9', '.', '+'), then pause voting, but continue to loop while adding the set of numerical characters to two temporary buffers (one for each file).
3. Continue to loop until both files have come to a non-numerical character. Convert the values in the two temporary buffers to real numbers and then compare them based on a predetermined "closeness" factor.
4. If the numbers match up, then resume the normal voting procedure.
5. Otherwise, exit the loop and return false.

Case-Sensitivity The case-sensitivity problem was handled simply by converting all 'a-z' or 'A-Z' characters that were encountered to their lowercase form before voting. This effectively provided a common standard for being able to compare the two temporary response files.

After integrating the algorithms to handle these special cases, various syntactic equivalence tests were performed on the system. The results of these tests and the system's successes and failures will be described in the following chapter.

Tagging

Tagging was also implemented in a manner similar to that described in the design phase. However, it was soon discovered that the PHP scripting language, while using the CGI interface, differed from CGI in that it did not automatically convert headers into environment variables, but rather stored them in a separate array of server variables. Hence, getting them into the environment required an intelligent, non-intrusive solution. The solution that was implemented basically involved a minor change to the PHP configuration. The default configuration was modified so as to always include a custom-made PHP script that would lookup the server variable corresponding to the custom tag header. This value was then forced into the environment by making a call to the PHP method "putenv()." By making this change, the framework for tagging was established and the only question that remained to be answered was whether this tag would then be able to be parsed out by a driver manager running in the same environment as the PHP process.

4.2.3 Transaction Mediator

There were two attempts made at developing an effective transaction mediator. The first one provided a more elegant solution, however, it proved to be quite complex to develop and also wasn't able to perform all of the actions that it was required to. As a result, a faster, less elegant solution was developed. Both implementations will be presented in the sections to follow. Figures-4-2 and 4-3 illustrate some of the structural differences between the two different implementations of the transaction mediator and show where they fit into the entire KARMA system.

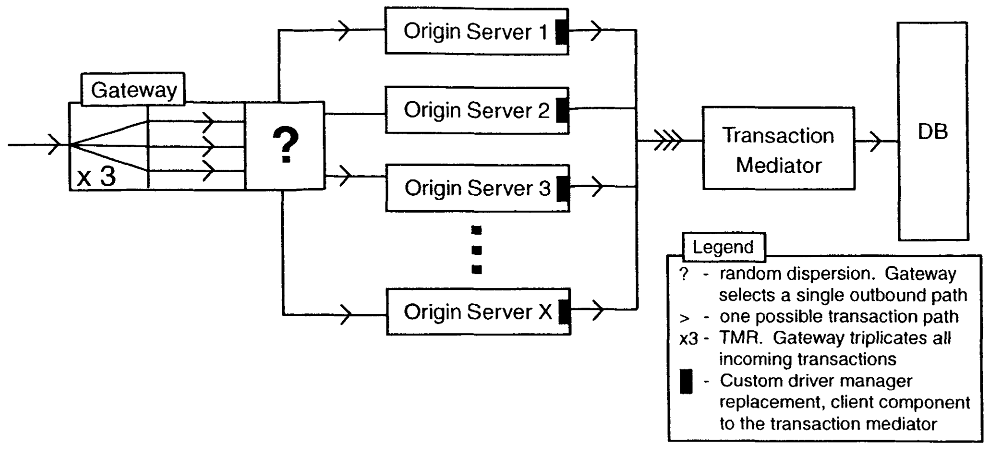

Gateway
x 3
?
Origin Server 1
Origin Server 2
Origin Server 3
Origin Server X
Transaction Mediator
DB
Legend
? - random dispersion. Gateway selects a single outbound path
> - one possible transaction path
x3 - TMR. Gateway triplicates all incoming transactions
▮ - Custom driver manager replacement, client component to the transaction mediator

Figure 4-2: Mediator One

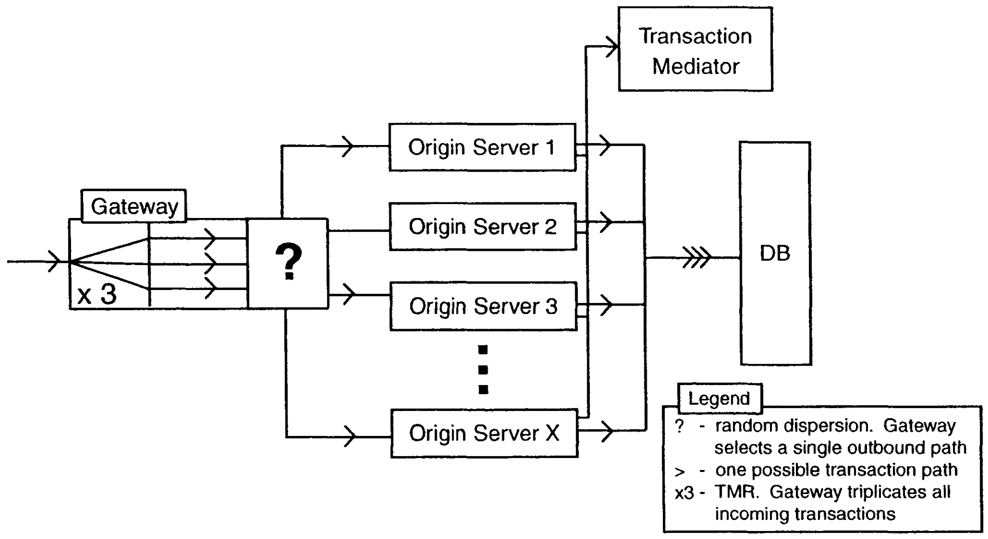

Transaction Mediator
Gateway
x 3
?
Origin Server 1
Origin Server 2
Origin Server 3
Origin Server X
DB
Legend
? - random dispersion. Gateway selects a single outbound path
> - one possible transaction path
x3 - TMR. Gateway triplicates all incoming transactions

Figure 4-3: Mediator Two

Mediator One - Initial Investigation

The first mediator consisted of two main parts: 1. a custom driver manager installed on each origin server, and 2. the transaction mediator itself. The custom driver manager was designed to act as a fully-functional replacement for the standard ODBC driver manager that was compiled in with PHP at installation. By installing a custom driver manager on each origin server, the transaction mediator could vote on the requests being made to the database, without a script ever being aware of its presence. As Figure-4-2 shows, the custom driver manager located on each origin server would be compiled in with PHP at installation. Then, for every subsequent script that made a request to the database, the custom driver manager would be called and it was then responsible for extracting a tag from the environment, and also for sending both the tag, and the original database request to the transaction mediator. The transaction mediator, upon receiving the first request would add this instance of the tag to its voting cache and then await the remaining requests. Upon receiving all three database requests, the mediator would then increment the request count, and as a result of the completed count, would proceed to vote upon the queries it had received.

Problems Encountered The problems with the initial mediator were more developmental than inherent in the design itself. However, it was revealing of the complexity of developing such a system. Because it required a custom driver manager to be compiled in with the PHP at installation, steps needed to be taken to mimic the functionality of a "true" driver manager. In actuality though, the custom driver manager was only mimicking a real driver manager, because it required the use of a transaction mediator to complete an entire transaction. Hence, the functionality of a "true" driver manager could only be achieved by merging the custom driver manager and the transaction mediator. Implementing a driver manager alone was no simple task. Implementing one that was sufficient to look like one and also portable enough to run on any of the variable OSs proposed by the KARMA system was even more difficult. Initially, there were a number of problems with byte-ordering across the different architectures. The list of bugs then went on to include problems with handling multiple connections, large data fields, and more. In the results section it will also be shown that the phase one transaction mediator suffered a tremendous decrease in performance. For these reasons, it seemed that a more lightweight solution might present a better solution for the simulation KARMA-2 system.

Mediator Two - A Simplified, Functional Version

Like the gateway, the second transaction mediator was implemented using the asynchronous socket package provided with SFS. The reason why this implementation of the transaction mediator was "less elegant" is because it required an additional modification to the COTS PHP configuration. In addition, it required a minor change to be made to every script that included a database request. The second phase transaction mediator involved adding to the custom PHP script that was loaded up by default to handle tagging. To the existing script file, a method was added that would essentially send a database request to the transaction mediator for approval. Then upon a successful vote, the mediator would send back the appropriate response to the method, which would then either make the actual database call, or would allow the call to be made by another one of the origin servers in its triplicated set. Making the actual database call was denied for some servers, even upon a successful vote, because if the request was an INSERT statement, for example, the request should only be executed once so as not to create multiple copies of a single entry within the database. A more detailed diagram of the mediator's operation is shown in Figure-4-4.

From Figure-4-4, it can be seen that the functionality of the mediator was much like a scaled down version of the gateway. Unlike the gateway, which acted as both a client and a server, the mediator was only responsible for acting as a server, and thus, the implementation was greatly simplified. Like the phase one mediator, the phase two system required the use of two components to complete the entire voting function of the mediator. The first component, as described earlier, was resident in a custom PHP script that was forced to load with every dynamic page. It was mentioned earlier that this PHP script would send a database request to the mediator for approval, however, additional details were not given as to the actual protocol for sending this data across. The actual format for sending this request for approval was as follows:

- Sending - The protocol for sending a request for approval followed very closely, the model for HTTP request headers. Each mediator request consisted of a set

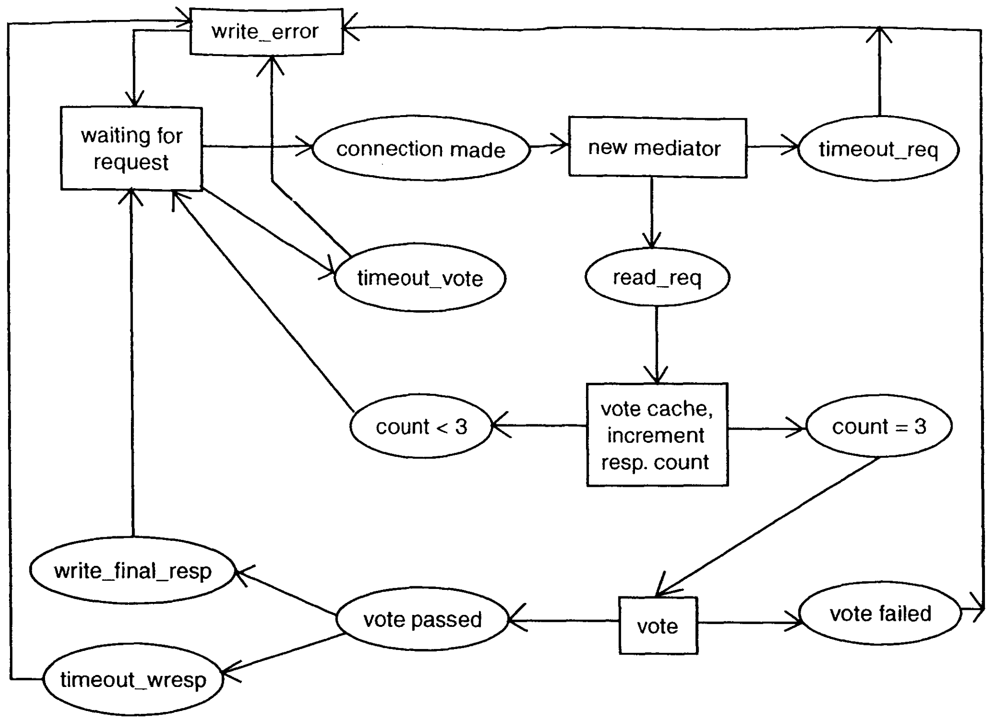


Figure 4-4: Mediator Block Diagram

of custom KARMA headers that could easily be parsed out by the mediator. The custom KARMA headers that were used are further defined below:

- Query-Id - This was the tag value used to uniquely identify a given query
- Query - This field would hold the actual database request
- Query-Type - This field would let the mediator know if the request was not a database altering statement (i.e. select), in which case the mediator could return true to all servers, allowing each to execute the database request. Otherwise, the mediator would return true to only one appointed server.

• Receiving - The response returned by a mediator was simply True or False. If the response was true, then the database query would be executed using the standard database calls, otherwise, the method would complete and the script would resume its normal execution.

As was stated earlier, the mediator component functioned much like a scaled down version of the gateway. In its idle state, it would simply await an incoming transaction. Upon receiving an initial request, it would then parse out the Query-Id, Query and Query-Type from the headers. The Query-Id value would then be placed in a voting cache along with a response count of one. After receiving the other two corresponding queries, the response count would reach the final state of three and would initiate the voting process. In the simulation stage, the voting process consisted solely of a string matching, requiring that all queries be exactly the same. Once voting had been completed successfully, the system would then check the Query-Type, and if the Query was a database altering request, the mediator would then send back the responses of True, False, False in no particular order. Likewise, if the Query was not a database altering request, i.e. SELECT, it would return True to all of the origin servers. Finally, if voting failed, the mediator would simply return False to all of the origin servers.

Tagging In the phase two system, the proposed tagging solution was unnecessary since the calls to the mediator were made from within PHP. Rather than requiring a clever manipulation of environment variables, the script was able to directly pass a tag, or Query-Id, to the mediator by simply reading the value of the appropriate server variable and setting it in the mediator request headers. One thing that was observed in implementing this solution was that, using the proposed solution for tagging, the environment variable method would inherently fail given a script with multiple database calls. This is because the mediator would then be unable to distinguish between the various database calls, since all would share the same environment/tag value, and might try to vote on the first request from one origin server against the second request from another.

Syntactic Equivalence In the simulation version of the KARMA-2 system, the problem of syntactic equivalence was implemented using a fully constrained query solution. However, if query flexibility was found to be truly vital to the security of the system, for all requests except for the SELECT statement, an enhanced equivalence algorithm could be implemented by using SQL objects, as were previously defined in Chapter 3: Design.

Problems Encountered

The second mediator implementation, while providing a more lightweight solution, also paid a significant price in regard to security. The most obvious shortcoming is in the fact that an intruder who successfully gained root privileges could simply modify a script such that it would call the standard database execute methods rather than the modified method that was used to initiate voting. Furthermore, even if additional security measures were taken to try and ensure that the standard database methods could not be called, these prevention methods would be PHP based, and could easily be bypassed by an intelligent intruder. Still, this vulnerability is not as bad as it sounds, in that the problem is non-existent in the real KARMA-2 system. Rather, this weakness only arises in the simulation system as a result of all the services being run on a single host. Due to the lack of distinction between the origin server and mediator machines, there is no ability for the database to be configured to only allow access from a single host (the mediator). Clearly, in a real world situation where only a trusted mediator would be configured with permissions to connect to the database it would be impossible to run such a simple bypass attack to disable voting at the mediator altogether. Hence, though this is a problem that exists in the simulation system, it will not be tested for attack since it can easily be prevented in the real KARMA system.

Chapter 5

Results

5.1 Attack Types

The KARMA-2 system was implemented to test the advantages of a system employing redundancy and using voting to detect intrusions and anomalies within the system. Furthermore, in order to perform the type of actions required to create such anomalies in the system, i.e. generating bad web pages, modifying database queries, it was assumed that an intruder would have to have gained the highest level of access privileges to the system, namely "superuser" status. As a result, the KARMA-2 system was designed to detect an attack on the system after an intruder had already compromised one of the origin servers. To simulate such an attack, one of the servers was modified in a variety of ways to generate a set of attacks for testing the robustness of the simulation KARMA-2 system.

The attack types again which the KARMA-2 system was tested were:

- Modified scripts - an attacker might modify a script so that it would return falsified HTML.

- Substituted webpages - an attacker might substitute a static page with one of their own. Similar to the above case.

- Modified database queries - an attacker would modify a script so that a malicious database command could be executed.

5.2 False Alarms

In addition to testing for possible attacks, a number of tests were performed to test the system's ability to bypass the common false alarm issues that were mentioned for voting at the gateway on outbound responses. These included the following:

- Numerical Rounding - two numbers representing the same value might vary in their precision, i.e. .099999 and .1.
- Whitespace/Carriage Return - any number of whitespace/carriage returns could appear between words.
- Case-Sensitivity - Words could be written in either upper or lower case.

5.3 Test Results

The tests were run by using a set of custom scripts and modifying them on a single origin server to exploit one of the above conditions (attack, false alarm). The results are shown in Table-5.1 (note that in all cases, a correct response page is still returned since voting should only fail with one server):

From the results, it is evident that voting is in fact possible and does work effectively on most of the cases that were presented in the design of the KARMA system. One problem with the simulation system that was quickly discovered was an incompatibility between the web servers used. It was found that AOLServer, which was used for the first origin server, returned a "text/html" response for PHP scripts, while the Apache servers returned responses in a chunked format. This caused some problems with the testing of voting results in that one server would always fail the vote for dynamic pages. This was also the reason for the "Y/N" response to the "modified script" test in Table-5.1. For that test, the voting was successful in detecting the one intrinsic error in addition to the character mismatch from the modified text, however, the system was unable to return an appropriate response since two rounds of voting failed. However, for static pages, the responses were returned from both servers in a

Test Type	Description	Expected Error	Success
Modified PHP	Modified the line “echo <table width=600 cellpadding=2>;” from display-movies.php to read “echo <table width=600>;”	Character mismatch error	Y/N
Modifed Static Page	Modified the line “<TITLE>Amenities In and Around Technology Square</TITLE>” from amenities.htm to read “<TITLE>Amenities Around Technology Square</TITLE>”	Character mismatch error	Y
Substituted Static Page	Replaced file location.htm with ownership.htm	Character mismatch error	Y
Modified Database Query (Select)	Changed the line “$sql=”select * from users order by username”;” from view_users.php to “$sql=”select * from users order by user_id”;”	Database error, Character mismatch on modified server’s response	Y
Modified Database Query (Update)	Changed the line “$sql = ”update genres set genre=’$newgenre’ where genre_id=$id”;” to “$sql = ”update genres set genre=’MyHackerGenre’ where genre_id=$id”;	Database Error, MyHackerGenre ignored and genre updated correctly	Y
Numerical Rounding	Created a new set of pages on the three origin servers containing a single line. os1’s test.html contained 1.000, while os2 and os3’s test.html contained .9999999”	.9999999 displayed to user, no errors	Y
Whitespace	Added a set of gratuitous whitespaces and carriage returns throughout the static page buildings.htm	No errors	Y
Whitespace in String	Changed the string “Technology Square Home” in the file restaurants.htm to “Technology Sq are Home”	Character mismatch error	Y
Whitespace in HTML	Changed the string “<TABLE BORDER=0 CELLSPACING=0” in the file leasing.htm to “<TABLE BO RDER=0 CELL SPACING=0”	Whitespace/CR error	Y
Case-Sensitivity	Modified the line “<TITLE>Campus News at Technology Square</TITLE>” in the page campusnews.htm to “<title>Campus News at Technology Square</tItLe>”	No errors	Y

Table 5.1: Test Results

text/html format. As a result, the intrinsic error was non-existent in these cases and as such, most of the tests were conducted using static pages.

5.4 Response Times

In addition to testing the feasibility of voting, one of the primary goals of the KARMA system was to design an intrusion detection system that would be efficient enough to perform its duties without affecting (significantly) the overall performance of the system.

The response times were tested using the following classes of pages:

- Static pages
- Scripts with minimal database access
- Scripts with a large number of database accesses

The algorithm used to calculate response times was implemented in the following manner. The UNIX *time* command was used to time a series of 50 connections to the KARMA system to retrieve a desired webpage. The total time for the 50 connections was then averaged to yield a final response time. The response times were measured for connecting directly to each of the individual origin servers and also for connecting via the KARMA system. A simple script was written to simplify the execution of the 50 consecutive connections. The script that was used is shown below:

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

int main(int argc, char **argv) {
  int port;
  char *page;
  char command[50];

  if (argc > 2) {
    port = (int)atoi(argv[1]);
    page = argv[2];
    sprintf(command, "wget http://localhost:%d/%s", port, page);

    for (int i = 0; i < 50; i++)
      system(command);

    // port is the port number of the desired server
    //     8000 for os1
    //     8001 for os2
    //     8002 for os3
    //     80 for the KARMA gateway
    // page is the desired webpage
  }
}
```

The above program was compiled and used to generate the executable, testpage. This binary was then used as follows to perform the final testing of response times:

```
bash% testpages <port> <page>
```

Each server was given three chances to yield its best total response time for 50 connections. The resulting best response times were tabulated and are shown below in Table-5.2:

Table 5.2: KARMA Response Times

Requested Page	OS1	OS2	OS3	KARMA
Static Page - amenities.htm	1.132s	1.603s	1.587s	1.819s
Script (min DB) - view_ratings.php	1.058s	2.890s	2.968s	17.442s
Script (max DB) - show_all_movies.php	1.017s	4.016s	3.953s	23.303s

It was surprising to see that the response time for the KARMA system was nearly six times slower than the original system. However, without further testing, it was difficult to tell which component in the KARMA system was most directly affecting this. Hence, to narrow down the source of the performance hit, one additional set of tests was performed while disabling voting at the mediator. The results of that test were as follows:

- Static Page - Irrelevant as no mediator voting would ever take place.

- Script (min DB) - 8.396s
- Script (max DB) - 12.925s

Once again, it was surprising to see that voting at the mediator contributed to a majority of the loss in response time even though the voting algorithm in the KARMA-2 system consisted solely of a string comparison. One possibility is that the times taken to connect and read data back and forth from the mediator were so inefficient as to degrade the overall performance of the voting at M_{in}.

Chapter 6

Conclusions

The KARMA-2 system was designed to efficiently detect intrusions and examine the possibilities of doing so by using random dispersion and voting. The results with respect to voting proved to be quite a success. By voting at the gateway on outbound responses and the mediator on inbound database requests, the system was able to effectively catch most of the proposed attacks on the KARMA system. Furthermore, the system was able to operate without triggering any significant number of false alarms. Much of this was due to the equivalence testing that was performed on the outbound responses and a number of constraints that were placed on the SQL calls being made to the database.

Unfortunately, the success of the KARMA system came at a slight cost in performance. The time taken to retrieve a set of common web pages was slowed down by almost a factor of six. Though the difference in time was only a matter of seconds, it is very probable that this factor would only get worse as the number of users and complexity of scripts continued to grow. It was indeed surprising to discover that the performance hit was as large as it was.

Despite this shortcoming, the KARMA system was successful in doing what it was designed to do. The simulation provided valuable verification that such a system could be built and further optimized to enhance performance with respect to response times. Though the KARMA system might not be guaranteed to catch every possible attack, it does provide an interesting alternative to many of the intrusion detection systems that are in use today. Furthermore, it provides a solution that is narrow enough in scope that it does not try and do "too much," but rather provides effective security for a limited set of Internet services – in this case, the worldwide web. What was most encouraging about the KARMA-2 simulation system was that its greatest weakness seemed to be in speed rather than security. Since speed could invariably be improved with more efficient algorithms and a more robustly programmed gateway/mediator, it is hopeful that the KARMA system might someday be implemented and used in the real world.

Chapter 7

Future Work

In spite of its success at detecting the attacks against which it was tested, there were a number of improvements that could have been made to the KARMA system. These improvements ranged from performance enhancing to security enhancing. Furthermore, a great deal of additional testing could have been done to verify the robustness and scalability of the system.

7.1 Performance Enhancements

As was shown in the results section the voting stages at the gateway and mediator caused a significant variation in the amount of time taken to receive a response from the origin servers. This could potentially be improved in a number of ways. One way would be to enhance the voting algorithms used. For example, it was mentioned earlier that MD5 hashes of static webpages could be cached to improve the speed of voting at G_{out}. This would improve the speed of voting by allowing the gateway to compute the hash of a single page and then vote on a set of short string rather than the length of an entire webpage.

The speed of the system could also be improved by utilizing faster disks, or by finding a better temporary storage location for the responses returned from the origin servers. As was mentioned in Chapter 5, the simulation system would write each response to disk while waiting for the remaining origin servers to return their respective responses. As a result, the costly read and write times for IDE storage could have contributed to the system's slower response times.

Lastly, the system's performance would probably have been improved if a more complete mediator was developed. As was mentioned earlier, the KARMA-2 simulation system was developed using a mediator that simply handled voting. Therefore, the mediator, rather than making requests directly to the database, would simply acknowledge a successful vote to each origin server which would then perform the database request on its own. As a result, every transaction would require approximately six connections, three to the mediator and then another three to the database. However, in a "more complete" KARMA system, the mediator would effectively be integrated into an emulator for the driver manager. Hence, only a single connection would need to be made to the database in addition to the three from the origin servers to the mediator. By requiring only four connections, the KARMA-2 system would most likely reduce the overhead required in establishing redundant connections to the database and also of transmitting large amounts of redundant data across the network.

7.2 Security Enhancements

In addition to these performance enhancements, there were a number of steps that could be taken to improve the security of the system. For the simulation system, only a small set of the most obvious attacks were performed against the system. However, given a malicious hacker with a great deal of time and knowledge, it would be almost impossible to predict the types of attacks that might be launched against the system. Hence, the system would most likely be improved by exposing it to either a real world attacker, or a dedicated team of attackers to thoroughly examine all of the vulnerabilities within the system.

7.3 General Enhancements

Finally, the system was slightly limited in that a compromise between complexity and flexibility was often necessary. The clearest example of this was seen in the case of SQL query voting. In order to make the system as simple as possible (for simulation) the query voting consisted of a single string comparison. However, if the system were to provide the greatest amount of flexibility for the developers and script writers, it would probably be better if the system could support out of order arguments, at the very least. This would require significant changes to the code though, and would also require the development of SQL objects that could then be used to determine the equivalence of two syntactically different but semantically equivalent SQL statements.

Clearly, this is not a comprehensive list of all the possible enhancements that could be made to the system. However, from the above examples, it is evident that there is still a great deal of work to be done. In spite of this, given the well-designed framework for the KARMA system and its success in detecting a preliminary set of attacks, the hopes for developing a new and effective intrusion detection system look very promising and not too far in the future.

Appendix A

Source Code

A.1 Gateway

A.1.1 webproxy.h

```
#include "http.h"
#include <stdio.h>

static const str VOTEFILE_PREFIX = "/tmp/resp_";

struct vote_entry *votecache;
struct vote_entry *vctail;

struct vote_entry {
  long fileid;
  int respcount;
  vote_entry *next;
  vote_entry *prev;

  vote_entry (long fid, int rc) {
    fileid = fid;
    respcount = rc;
    next = NULL;
    prev = NULL;
  }

  void insert (vote_entry *ve) {
    ve->next = NULL;

    if (vctail) {
      vctail->next = ve;
      ve->prev = vctail;
      vctail = ve;
    } else {
      ve->prev = NULL;
      vctail = ve;
      votecache = ve;
    }
  }

  void remove (vote_entry *ve) {
    if (vctail == votecache) {
      vctail = NULL;
      votecache = NULL;
    } else if (ve == votecache) {
      votecache = ve->next;
      votecache->prev = NULL;
    } else if (ve == vctail) {
      vctail = ve->prev;
      vctail->next = NULL;
    } else {
      ve->prev->next = ve->next;
      ve->next->prev = ve->prev;
    }
  }

  vote_entry *find (int fid) {
    vote_entry *tmp = vctail;

    while (tmp) {
      if (tmp->fileid == fid) {
        return tmp;
      } else {
        tmp = tmp->prev;
      }
    }
    return NULL;
  }

};

struct connection {
  str host;
  int port;
  int request_id;
int vote_id;

connection()
  : port(-1), request_id(-1), vote_id(-1), deof(false),
       delay_req(NULL), delay_rresp(NULL), dsock(-1),
       fcreated(false), gotresp(false) {
  host = "localhost";
  htresp = New httpresp();
  htreq = New httpreq();
}

connection(str h, int p, long rid, int vid)
  : deof(false), delay_req(NULL), delay_rresp(NULL),
       dsock(-1), fcreated(false), gotresp(false) {
  host = h;
  port = p;
  request_id = rid;
  vote_id = vid;
  htresp = New httpresp();
  htreq = New httpreq();
  htresp->request_id = rid;
  htreq->request_id = rid;
}

~connection() {
  if (delay_req) {
    timecb_remove(delay_req);
    delay_req = NULL;
  }

  if (delay_rresp) {
    timecb_remove(delay_rresp);
    delay_rresp = NULL;
  }

  // delete the associated file
  remove(get_filename());

  if (dsock >= 0) {
    fdcb (dsock, selread, NULL);
    fdcb (dsock, selwrite, NULL);
    close (dsock);
  }
}
```

```
    str get_filename() {
      strbuf filebuf;
      filebuf << VOTEFILE_PREFIX << request_id << "." << vote_id;
      str fname = filebuf;
      return fname;
    }

    httpreq *htreq;
    httpresp *htresp;
    bool deof;
    timecb_t *delay_req;
    timecb_t *delay_rresp;
    int dsock;
    bool fcreated;
    bool gotresp;
    strbuf resp_buf;
  };

  struct webproxy {
    connection* connections[3];
    httpreq *htreq;
    httpresp *htresp;
    bool gotreq;
    bool gotresp;

    webproxy();
    ~webproxy();
    int conn_id;

    void timeout_vote(long rid);
    void timeout_wresp();
    void timeout_req(connection *cn);
    void timeout_rresp(connection *cn);

    void err_handle(int type, connection *cn, int fd);
    void connected(connection *cn, int sfd);
    void duplicate_request();
    void make_requests();
    void read_resp(connection *cn);
    void read_req();
    void write_resp(connection *cn);
    void write_req(connection *cn);
    void write_error(str errmsg);
    void write_final_resp(int rfd);
    void write_error_resp();
    void vote(long rid);

    timecb_t *delay_vote;
    timecb_t *delay_wresp;

    int osock;
    strbuf req_buf, resp_buf;
    bool oeof, veof;
  };
```

A.1.2 webproxy.C

```
#include "async.h"
#include "webproxy.h"

#include <fcntl.h>
#include <string.h>
#include <fstream.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>

static const unsigned int BUF_MAX = 2048;
int listensock;
int listenport;
enum { EXIT_ERR, EOF_ERR };

// function prototypes
void new_connection();

int request_id; /* unique id for each http_request
                   if Connection: close does what we think
                   then it is possible that request_id will
                   be the same as curid */

int main(int argc, char **argv) {
  if (argc == 2) {
    listenport = (int)atoi(argv[1]);
  } else if (argc > 2) {
    printf("Usage: ./webproxy <port>\nport = \tPort to
       listen on.\n\tDefaults to 80 if none specified.\n");
    exit(1);
  } else {
    listenport = 80;
  }

  votecache = NULL;
  vctail = NULL;

  request_id = 0;
  setprogname(argv[0]);
  listensock = inetsocket(SOCK_STREAM, listenport);

  if (listensock < 0) {
    fatal << "Error creating listener socket.\n";
    exit(1);
  }

  make_async(listensock);
  listen(listensock, 5);
  fdcb(listensock, selread, wrap(&new_connection));
  amain();

  return 0;
}

void new_connection() {
  request_id++;

  vNew webproxy();
}

webproxy::webproxy()
  : gotreq(false), gotresp(false), delay_vote(NULL),
      delay_wresp(NULL), osock(-1), oeof(false),
      veof(false) {
  htreq = New httpreq();
  htresp = New httpresp();
  htreq->request_id = request_id;
  conn_id = request_id;
  sockaddr_in sin;
  bzero(&sin, sizeof(sin));
  socklen_t sinlen = sizeof(sin);

  osock = accept(listensock,
     reinterpret_cast<sockaddr *>(&sin), &sinlen);
  if (osock < 0) {
    delete this;
    return;
  }

  fdcb(osock, selread, wrap(this, &webproxy::read_req));
}

void webproxy::duplicate_request() {
  connections[0]->htreq = htreq->clone();
  connections[1]->htreq = htreq->clone();
  connections[2]->htreq = htreq->clone();
}
```

```
void webproxy::make_requests() {
  str hname = "127.0.0.1";

  connection *connone = New connection(hname, 8000,
     request_id, 1);
  connection *conntwo = New connection(hname, 8001,
     request_id, 2);
  connection *connthree = New connection(hname, 8002,
     request_id, 3);

  connections[0] = connone;
  connections[1] = conntwo;
  connections[2] = connthree;

  duplicate_request();
  delay_vote = delaycb(30, 0, wrap(this,
     &webproxy::timeout_vote, request_id));

  tcpconnect("127.0.0.1", 8000, wrap(this,
     &webproxy::connected, connone));
  tcpconnect("127.0.0.1", 8001, wrap(this,
     &webproxy::connected, conntwo));
  tcpconnect("127.0.0.1", 8002, wrap(this,
     &webproxy::connected, connthree));
}

void webproxy::connected(connection *cn, int sfd) {
  if (sfd < 0) {
    write_error("Failed to connect to server.");
    return;
  }

  cn->dsock = sfd;

  cn->delay_req = delaycb(10, 0, wrap(this,
     &webproxy::timeout_req, cn));
  fdcb(cn->dsock, selwrite, wrap(this,
     &webproxy::write_req, cn));
}

void webproxy::read_req() {
  switch(req_buf.tosuio()->input(osock)) {
  case -1:
    if (errno != EAGAIN) {
      fdcb(osock, selread, NULL);
      delete this;
    }
    break;
  case 0:
    fdcb(osock, selread, NULL);
    err_handle(EOF_ERR, NULL, osock);
    break;
  default:
    if (!gotreq) {
      int pval = htreq->parse(req_buf.tosuio());
      if (pval == 1) {
        gotreq = true;
        htreq->headers << "\r\n";

        htreq->headers.tosuio()->take(req_buf.tosuio());
        make_requests();
      }
    } else {
      htreq->headers.tosuio()->take(req_buf.tosuio());
      duplicate_request();
    }
  }
}

void webproxy::write_req(connection *cn) {
  if (cn->delay_req) {
    timecb_remove(cn->delay_req);
    cn->delay_req = NULL;
  }

  switch(cn->htreq->headers.tosuio()->output(cn->dsock)) {
  case 1:
    if (cn->htreq->headers.tosuio()->resid() <= 0) {
      fdcb (cn->dsock, selwrite, NULL);
      cn->delay_rresp = delaycb(10, 0, wrap(this,
         &webproxy::timeout_rresp, cn));
      fdcb(cn->dsock, selread, wrap(this,
         &webproxy::read_resp, cn));
    }
    break;
  case -1:
    delete this;
    break;
  default:
      break;
    }
  }

  void webproxy::read_resp(connection *cn) {
    vote_entry *ve;

    if (cn->delay_rresp) {
      timecb_remove(cn->delay_rresp);
      cn->delay_rresp = NULL;
    }

    if (cn->resp_buf.tosuio()->resid() > BUF_MAX) {
      fdcb(cn->dsock, selread, NULL);
    }

    switch(cn->resp_buf.tosuio()->input(cn->dsock)) {
    case -1:
      if (errno != EAGAIN) {
        fdcb(cn->dsock, selread, NULL);
        delete this;
      }
      break;
    case 0:
      fdcb(cn->dsock, selread, NULL);
      err_handle(EOF_ERR, cn, cn->dsock);

      ve = votecache->find(cn->request_id);

      if (ve) {
        ve->fileid = cn->request_id;
        ve->respcount = ve->respcount+1;

      } else {
        ve = New vote_entry(cn->request_id, 1);
        votecache->insert(ve);
      }

      if (ve->respcount == 3) {
        if (delay_vote) {
          timecb_remove(delay_vote);
          delay_vote = NULL;
        }
        votecache->remove(ve);
        vote(cn->request_id);
    }

    break;
  default:
    if (!cn->gotresp) {
      int pval = cn->htresp->parse(cn->resp_buf.tosuio());

      if (pval == 1) {
        cn->gotresp = true;
        cn->htresp->headers << "\r\n";
      }
    }

    if (cn->gotresp) {
      cn->htresp->headers.tosuio()->take(
         cn->resp_buf.tosuio());
      write_resp(cn);
    }
    break;
  }
}

void webproxy::write_resp(connection *cn) {
  int rfd; //response file descriptor

  str fname = cn->get_filename();

  if (!cn->fcreated) {
    remove(fname);
    rfd = open(fname, O_CREAT+O_WRONLY+O_APPEND, 0666);
  } else {
    rfd = open(fname, O_WRONLY+O_APPEND);
  }

  if (rfd == -1) { //if other error.
    perror(fname);
    return;
  } else {
    cn->fcreated = true;
  }

  switch(cn->htresp->headers.tosuio()->output(rfd)) {
  case 1:
    if (cn->htresp->headers.tosuio()->resid() <= 0) {
      fdcb (cn->dsock, selwrite, NULL);
      cn->delay_rresp = delaycb(10, 0, wrap(this,
         &webproxy::timeout_rresp, cn));
      fdcb (cn->dsock, selread, wrap(this,
         &webproxy::read_resp, cn));
    }
    break;
  case -1:
    delete this;
    break;
  default:
    break;
  }

  close(rfd);
}

void webproxy::vote(long rid) {
  strbuf file_one, file_two, file_three;

  file_one << VOTEFILE_PREFIX << rid << ".1";
  file_two << VOTEFILE_PREFIX << rid << ".2";
  file_three << VOTEFILE_PREFIX << rid << ".3";

  str fone = file_one;
  str ftwo = file_two;
  str fthree = file_three;

  int rfd;

  // if any two match.
  if (compare_files(fone, ftwo)) {
    rfd = open(fone, O_RDONLY);

    if (rfd >= 0) {
      delay_wresp = delaycb (10, 0, wrap (this,
         &webproxy::timeout_wresp));
      fdcb(osock, selread, NULL);
      fdcb(osock, selwrite, wrap(this,
         &webproxy::write_final_resp, rfd));
      return;
    }
  }

  if (compare_files(ftwo, fthree)) {
    rfd = open(ftwo, O_RDONLY);
    if (rfd >= 0) {
      delay_wresp = delaycb (10, 0, wrap (this,
         &webproxy::timeout_wresp));
      fdcb(osock, selread, NULL);
      fdcb(osock, selwrite, wrap(this,
         &webproxy::write_final_resp, rfd));
      return;
    }
  }

  if (compare_files(fone, fthree)) {
    rfd = open(fone, O_RDONLY);

    if (rfd >= 0) {
      delay_wresp = delaycb (10, 0, wrap (this,
         &webproxy::timeout_wresp));
      fdcb(osock, selread, NULL);
      fdcb(osock, selwrite, wrap(this,
         &webproxy::write_final_resp, rfd));
      return;
    }
  }

  write_error("Voting failed for all servers.");
}

void webproxy::write_final_resp(int rfd) {

  if (delay_wresp) {
    timecb_remove(delay_wresp);
    delay_wresp = NULL;
  }

  switch(resp_buf.tosuio()->input(rfd)) {
  case -1:
    if (errno != EAGAIN) {
      close(rfd);
      write_error("Invalid response received from server.");
    }
    break;
  case 0:
    close(rfd);
    veof = true;
    shutdown(osock, SHUT_WR);
    delete this;
    break;
  default:
    if (!gotresp) {
      int pval = htresp->parse(resp_buf.tosuio());

      if (pval == 1) {
        gotresp = true;
        htresp->headers << "\r\n";
      }
    }

    if (gotresp) {
      htresp->headers.tosuio()->take(resp_buf.tosuio());

      switch(htresp->headers.tosuio()->output(osock)) {
      case 1:
        break;
      case -1:
        delete this;
        break;
      default:
        break;
      }
    }
    break;
  }
}

void webproxy::write_error_resp() {
  switch(htresp->headers.tosuio()->output(osock)) {
  case 1:
    delete this;
    break;
  case -1:
    delete this;
    break;
  default:
    break;
  }
}

void webproxy::write_error(str errmsg) {
  suio buf_resp;
```

```
      oeof = true;
      for (int i=0; i<3; i++) {
        connection *conn = connections[i];
        shutdown(conn->dsock, SHUT_WR);
      }
    } else {
      cn->deof = true;
    }
  }

  if (veof && oeof && connections[0]->deof
      && connections[1]->deof
      && connections[2]->deof) {
    delete this;
  }
}

webproxy::~webproxy() {
  if (delay_vote) {
    timecb_remove(delay_vote);
    delay_vote = NULL;
  }

  if (delay_wresp) {
    timecb_remove(delay_wresp);
    delay_wresp = NULL;
  }

  if (osock >= 0) {
    fdcb (osock, selread, NULL);
    fdcb (osock, selwrite, NULL);
    close (osock);
  }

  delete connections[0];
  delete connections[1];
  delete connections[2];
}
```

```
  suio_print(&buf_resp, httperror(503, errmsg, htreq->url,
     errmsg));

  htresp->headers.tosuio()->take(&buf_resp);

  fdcb(osock, selread, NULL);
  fdcb(osock, selwrite, wrap(this,
     &webproxy::write_error_resp));
}

void webproxy::timeout_vote(long rid) {
  vote_entry *ve = votecache->find(rid);

  veof = true;
  delay_vote = NULL;
  if (ve->respcount < 3) {
    votecache->remove(ve);
    write_error("Server timed out while voting.");
  }
}

void webproxy::timeout_wresp () {
  fdcb(osock, selwrite, NULL);
  delay_wresp = NULL;
  err_handle(EXIT_ERR, NULL, osock);
}

void webproxy::timeout_req (connection *cn) {
  fdcb(cn->dsock, selwrite, NULL);
  cn->delay_req = NULL;
  err_handle(EXIT_ERR, cn, cn->dsock);
}

void webproxy::timeout_rresp (connection *cn) {
  fdcb(cn->dsock, selread, NULL);
  cn->delay_rresp = NULL;
  err_handle(EXIT_ERR, cn, cn->dsock);
}

void webproxy::err_handle(int type,
   connection *cn, int fd) {
  if (type == EXIT_ERR) {
    delete this;
  } else {
    if (fd == osock) {
```

A.2 Mediator

A.2.1 mediator.h

```
#include "http.h"
#include <stdio.h>

struct vote_entry *votecache;
struct vote_entry *vctail;

struct mediator {
  httpreq *htreq;
  bool gotreq;

  mediator();
  ~mediator();
  int conn_id;

  void timeout_vote();
  void timeout_wresp();

  void err_handle(int type, int fd);
  void read_req();
  void write_error(str errmsg);
  void write_error_resp(str errmsg);

  timecb_t *delay_vote;
  timecb_t *delay_wresp;

  int osock;
  bool oeof, veof;
  strbuf req_buf;
};

struct vote_entry {
  long queryid;
  int respcount;
  struct mediator *connections[3];
  vote_entry *next;
  vote_entry *prev;

  vote_entry (long qid, int rc, mediator *m) {
    queryid = qid;
    respcount = rc;
    connections[0] = m;
  connections[1] = NULL;
  connections[2] = NULL;
  next = NULL;
  prev = NULL;
}

void insert (vote_entry *ve) {
  ve->next = NULL;

  if (vctail) {
    vctail->next = ve;
    ve->prev = vctail;
    vctail = ve;
  } else {
    ve->prev = NULL;
    vctail = ve;
    votecache = ve;
  }
}

void remove (vote_entry *ve) {
  if (vctail == votecache) {
    vctail = NULL;
    votecache = NULL;
  } else if (ve == votecache) {
    votecache = ve->next;
    votecache->prev = NULL;
  } else if (ve == vctail) {
    vctail = ve->prev;
    vctail->next = NULL;
  } else {
    ve->prev->next = ve->next;
    ve->next->prev = ve->prev;
  }
}

vote_entry *find (int qid) {
  vote_entry *tmp = vctail;

  while (tmp) {
    if (tmp->queryid == qid) {
      return tmp;
    } else {
      tmp = tmp->prev;
    }
    }
    return NULL;
  }
};
```

A.2.2 mediator.C

```
#include "async.h"
#include "mediator.h"

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

static const unsigned int BUF_MAX = 2048;
int listensock;
int listenport;
enum { EXIT_ERR, EOF_ERR };

int request_id;

bool compare_files(str qone, str qtwo) {
  if (qone) {
    if (qtwo) {
      if (qone == qtwo)
        return true;
      else
        return false;
    }
    return false;
  }
  return false;
}

// function prototypes
void new_connection();
void vote(int qid);
void write_duplicate_errors(int qid, int prefid);

int main(int argc, char **argv) {
  if (argc == 2) {
    listenport = (int)atoi(argv[1]);
  } else if (argc > 2) {
    printf("Usage: ./mediator <port>\nport = \tPort to
       listen on.\n\tDefaults to 5555 if none specified.\n");
    exit(1);
  } else {
    listenport = 5555;
  }

  votecache = NULL;
  vctail = NULL;

  request_id = 0;
  setprogname(argv[0]);
  listensock = inetsocket(SOCK_STREAM, listenport);

  if (listensock < 0) {
    fatal << "Error creating listener socket.\n";
    exit(1);
  }

  make_async(listensock);
  listen(listensock, 5);
  fdcb(listensock, selread, wrap(&new_connection));
  amain();

  return 0;
}

void new_connection() {
  request_id++;

  vNew mediator();
}

mediator::mediator()
  : gotreq(false), delay_vote(NULL), delay_wresp(NULL),
       osock(-1), oeof(false), veof(false) {

  htreq = New httpreq();
  conn_id = request_id;
  sockaddr_in sin;
  bzero(&sin, sizeof(sin));
  socklen_t sinlen = sizeof(sin);

  osock = accept(listensock,
     reinterpret_cast<sockaddr *>(&sin), &sinlen);
  if (osock < 0) {
    delete this;
    return;
  }

  fdcb(osock, selread, wrap(this, &mediator::read_req));
}
```

```
void mediator::read_req() {
  switch(req_buf.tosuio()->input(osock)) {
  case -1:
    if (errno != EAGAIN) {
      fdcb(osock, selread, NULL);

      delete this;
    }
    break;
  case 0:
    fdcb(osock, selread, NULL);
    err_handle(EOF_ERR, osock);
    break;
  default:
    if (!gotreq) {
      int pval = htreq->parse(req_buf.tosuio());
      if (pval == 1) {
        gotreq = true;
        htreq->headers << "\r\n";

        htreq->headers.tosuio()->take(req_buf.tosuio());
        fdcb(osock, selread, NULL);

        vote_entry *ve;
        ve = votecache->find(htreq->queryid);

        if (ve && (ve->respcount != 0)) {
          ve->queryid = htreq->queryid;
          ve->respcount = ve->respcount+1;
          ve->connections[ve->respcount-1] = this;
        } else {
          delay_vote = delaycb(10, 0, wrap(this,
             &mediator::timeout_vote));
          ve = New vote_entry(htreq->queryid, 1, this);
          votecache->insert(ve);
        }

        if (ve->respcount == 3) {
          if (delay_vote) {
            timecb_remove(delay_vote);
            delay_vote = NULL;
          }
          vote(htreq->queryid);
        }
      }
    }
  }
}

void write_duplicate_errors(int qid, int prefid) {
  vote_entry *ve;

  ve = votecache->find(qid);
  mediator *m1 = ve->connections[0];
  mediator *m2 = ve->connections[1];
  mediator *m3 = ve->connections[2];
  str falsemsg;

  if (m1->htreq->querytype) {
    if (strncmp(m1->htreq->querytype, "select", 6) == 0) {
      falsemsg = "True";
    } else {
      falsemsg = "False";
    }
  } else {
    falsemsg = "False";
  }

  switch (prefid) {
  case -1:
    m1->write_error("False");
    m2->write_error("False");
    m3->write_error("False");
    break;
  case 1:
    m1->write_error("True");
    m2->write_error(falsemsg);
    m3->write_error(falsemsg);
    break;
  case 2:
    m1->write_error(falsemsg);
    m2->write_error("True");
    m3->write_error(falsemsg);
    break;
  case 3:
    m1->write_error(falsemsg);
    m2->write_error(falsemsg);
    m3->write_error("True");
    break;
  default:
    m1->write_error("False");
    m2->write_error("False");
    m3->write_error("False");
    break;
  }
  votecache->remove(ve);
}

void vote(int qid) {
  vote_entry *ve;

  ve = votecache->find(qid);
  mediator *m1 = ve->connections[0];
  mediator *m2 = ve->connections[1];
  mediator *m3 = ve->connections[2];

  // if any two match.
  if (compare_files(m1->htreq->query, m2->htreq->query)) {
    write_duplicate_errors(qid, 1);
    return;
  }

  if (compare_files(m2->htreq->query, m3->htreq->query)) {
    write_duplicate_errors(qid, 2);
    return;
  }

  if (compare_files(m1->htreq->query, m3->htreq->query)) {
    write_duplicate_errors(qid, 3);
    return;
  }

  write_duplicate_errors(qid, -1);
}

void mediator::write_error_resp(str errmsg) {
  strbuf resp_buf;

  resp_buf << errmsg << "\r\n\r\n";
  switch(resp_buf.tosuio()->output(osock)) {
  case 1:
    delete this;
    break;
  case -1:
    delete this;
    break;
  default:
    break;
  }
}

void mediator::write_error(str errmsg) {
  delay_wresp = delaycb (10, 0, wrap (this,
      &mediator::timeout_wresp));
  fdcb(osock, selwrite, wrap(this,
      &mediator::write_error_resp, errmsg));
}

void mediator::timeout_vote() {
  vote_entry *ve = votecache->find(htreq->queryid);
  veof = true;
  delay_vote = NULL;

  mediator *tmp;
  for (int i=0; i<3; i++) {
    tmp = ve->connections[i];
    if (tmp) {
      tmp->write_error("False");
    }
  }
  votecache->remove(ve);
}

void mediator::timeout_wresp () {
  fdcb(osock, selwrite, NULL);
  delay_wresp = NULL;
  err_handle(EXIT_ERR, osock);
}

void mediator::err_handle(int type, int fd) {
  if (type == EXIT_ERR) {
    delete this;
  } else {
    if (fd == osock) {
      shutdown(osock, SHUT_WR);
      oeof = true;
    }
  }

  if (veof && oeof) {
    delete this;
  }
}

mediator::~mediator() {
  if (delay_vote) {
    timecb_remove(delay_vote);
    delay_vote = NULL;
  }

  if (delay_wresp) {
    timecb_remove(delay_wresp);
    delay_wresp = NULL;
  }

  if (osock >= 0) {
    fdcb (osock, selread, NULL);
    fdcb (osock, selwrite, NULL);
    close (osock);
  }
}
```

A.3 G_{out} Voting Equivalence Algorithm

```
bool isNumber(char inch) {
  if ((inch == '1') || (inch == '2') || (inch == '3')
     || (inch == '4') || (inch == '5') || (inch == '6')
     || (inch == '7') || (inch == '8') || (inch == '9')
     || (inch == '0'))
    return true;
  else
    return false;
}

bool compare_files(str fileone, str filetwo) {
  ifstream infile(fileone);
  ifstream difffile(filetwo);

  filebuf *inbuf = infile.rdbuf();
  filebuf *diffbuf = difffile.rdbuf();

  char inch;
  char diffch;
  bool idquotes = false;
  bool isquotes = false;
  bool ddquotes = false;
  bool dsquotes = false;

  double inum;
  double dnum;
  bool ibypass;
  bool dbypass;
  bool numcomp;
  bool ichunk;
  bool dchunk;
  int precision = 10;
  int decimal, sign;
  bool igotdec, dgotdec;
  int idecpos, ddecpos;

  bool ijustnew = false;
  bool djustnew = false;

  // get past headers first
  do {
    inch = inbuf->snextc();
    if (inch == '\n')
```

```
      if (ijustnew)
        break;
      else
        ijustnew = true;
    else
      if (inch != '\r')
        ijustnew = false;
  } while ((inbuf->sgetc() != EOF));

  do {
    diffch = diffbuf->snextc();
    if (diffch == '\n')
      if (djustnew)
        break;
      else
        djustnew = true;
    else
      if (diffch != '\r')
        djustnew = false;
  } while ((diffbuf->sgetc() != EOF));

  do {
    inch = inbuf->sgetc();
    diffch = diffbuf->sgetc();

    inum = 0;
    dnum = 0;
    ibypass = false;
    dbypass = false;
    numcomp = false;
    ichunk = false;
    dchunk = false;
    igotdec = false;
    dgotdec = false;

    idecpos = 0;
    ddecpos = 0;

    // whitespace, LFCR check
    while (!idquotes && !isquotes && (inch != EOF) &&
        ((inch == ' ') || (inch == '\r') ||
         (inch == '\n') || (inch == '\t'))) {
      inch = inbuf->snextc();
      ichunk = true;
    }
```

```
while (!ddquotes && !dsquotes && (diffch != EOF) &&
   ((diffch == ' ') || (diffch == '\r') ||
    (diffch == '\n') || (diffch == '\t'))) {
  diffch = diffbuf->snextc();
  dchunk = true;
}

if ((ichunk && !dchunk) || (!ichunk && dchunk)) {
  warn << "Whitespace/CR mismatch.\n";
  return false;
}

// check for num
if (inch == '.') {
  igotdec = true;
  inch = inbuf->snextc();
  if (!isNumber(inch)) {
    inbuf->sungetc();
    ibypass = true;
  }
}

if (diffch == '.') {
  dgotdec = true;
  diffch = diffbuf->snextc();
  if (!isNumber(diffch)) {
    diffbuf->sungetc();
    dbypass = true;
  }
}

while (!ibypass && (inch != EOF) &&
       (isNumber(inch) || (inch == '.'))) {
  if (inch == '.') {
    if (igotdec) {
      inbuf->sungetc();
      break;
    } else {
      igotdec = true;
    }
  } else {
    if (igotdec)
      idecpos++;
    inum = (inum*10)+(atoi(&inch));
  }

  inch = inbuf->snextc();
  numcomp = true;
}

while (!dbypass && (diffch != EOF) &&
       (isNumber(diffch) || (diffch == '.'))) {
  if (diffch == '.') {
    if (dgotdec) {
      diffbuf->sungetc();
      break;
    } else {
      dgotdec = true;
    }
  } else {
    if (dgotdec)
      ddecpos++;

    dnum = (dnum*10)+(atoi(&diffch));
  }

  diffch = diffbuf->snextc();
  numcomp = true;
}

// quote check
if ((inch == '\'') && (!idquotes)) {
  isquotes = !isquotes;
} else if ((inch == '\"') && (!isquotes)) {
  idquotes = !idquotes;
}

if ((diffch == '\'') && (!ddquotes)) {
  dsquotes = !dsquotes;
} else if ((diffch == '\"') && (!dsquotes)) {
  ddquotes = !ddquotes;
}

// output
if (numcomp) {
  inum = inum*pow(inum, 0-idecpos);
  dnum = dnum*pow(dnum, 0-ddecpos);
```

```
    double diff;

    if (inum > dnum)
      diff = inum - dnum;
    else
      diff = dnum - inum;

    if (diff > .000001) {
      str dstr = fcvt(dnum, precision, &decimal,
         &sign);
      str istr = fcvt(inum, precision, &decimal,
         &sign);

      warn << "Numerical value mismatch.\n";
      return false;
    }

    inbuf->sungetc();
    diffbuf->sungetc();
  } else {
    if (inch != diffch) {
      // case-sensitivity check
      if ((inch >= 65) && (inch <= 90)) {
        if ((inch+32) != diffch) {
          warn << "Character mismatch.\n";
          return false;
        }
      } else if ((diffch >= 65) && (diffch <= 90)) {
        if ((diffch+32) != inch) {
          warn << "Character mismatch.\n";
          return false;
        }
      } else {
        warn << "Character mismatch.\n";
        return false;
      }
    }
  }

  inbuf->snextc();
  diffbuf->snextc();
} while ((inbuf->sgetc()!=EOF) &&
         (diffbuf->sgetc()!=EOF)) ;

if (inbuf->sgetc() != diffbuf->sgetc()) {
      return false;
    } else {
      return true;
    }
  }
```

Bibliography

[1] Dr. Myron L. Cramer, James Cannady, and Jay Harrell. New Methods of Intrusion Detection using Control-Loop Measurement. `http://www.infowar.com/survey/ids_newm.html`, May 1996.

[2] A.J. Flavell. ISO-8859-1 and the Mac Platform. `http://ppewww.ph.gla.ac.uk/~flavell/iso8859/iso8859-mac.html`, 2002.

[3] National Center for Supercomputing Applications. CGI Environment Variables. `http://hoohoo.ncsa.uiuc.edu/cgi/env.html`, 2002.

[4] Robert Graham. FAQ Network Intrusion Detection Systems. `http://www.ticm.com/kb/faq/idsfaq.html`, March 2000.

[5] The PostgreSQL Global Development Group. PostgreSQL 7.3devel Reference Manual. `http://developer.postgresql.org/docs/postgres/reference.html`, 2001.

[6] B. Hardekopf, K. Kwiat, and S. Upadhyaya. A Decentralized Voting Algorithm for Increasing Dependability in Distributed Systems. `http://www.cs.buffalo.edu/~shambhu/resume/sci2001.pdf`, 2001.

[7] Fred Kerby, Steven Moore, and Tim Aldrich. Intrusion Detection FAQ. `http://www.sans.org/newlook/resources/IDFAQ/ID_FAQ.htm`, 2002.

[8] Jonathan Korba. Windows NT Attacks for the Evaluation of Intrusion Detection Systems. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2000.

[9] Brian Krebs. Businesses Loath To Report Hack Attacks To Feds - FBI. `http://www.newsbytes.com/news/02/175718.html`, April 2002.

[10] D. Mazieres. SFS 0.5 Manual. `http://www.fs.net/sfs/new-york.lcs.mit.edu:85xq6pznt4mgfvj4mb23x6b8adak%55ue/pub/sfswww/sfs.ps.gz`, 1999.

[11] Kevin E. McDonald. A Lightweight Real-time Host-based Intrusion Detection System. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2001.

[12] Rolf Oppliger. *Security Technologies for the World Wide Web.* Computer Security. Artech House, Inc., Norwood, MA, 2000.

[13] Charles P. Pfleeger. *Security In Computing.* Prentice-Hall, Inc., Upper Saddle River, NJ, second edition, 1997.

[14] David A. Rennels. Fault-Tolerant Computing. `http://www.cs.ucla.edu/~rennels/article98.pdf`, 1998.

[15] Tom Robinson. The ISO 8859-1 Character Set. `http://ppewww.ph.gla.ac.uk/~flavel/iso8859/iso8859-mac.html`.

[16] Marcus D. Rosenbaum, Drew Altman, and Robert J. Blendon. Survey Shows Widespread Enthusiasm for High Technology. `http://www.npr.org/programs/specials/poll/technology/`, December 1999.

[17] William Stallings. *Network and Internetwork Security: Principles and Practice*, chapter 6, pages 207–237. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1995.

[18] Inc. Unicode. What Is Unicode? `http://www.unicode.org/unicode/standard/WhatIsUnicode.html`, 2002.

[19] W3C. The HTML 4.01 Specification. `http://www.w3.org/TR/REC-html40/charset.html`, December 1999.

[20] R. Wang, F. Wang, and G.T. Byrd. SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Server. `http://www.anr.mcnc.org/projects/SITAR/papers/smc01sitar.ps.gz`, 2001.

[21] Team Web. Special Characters in HTML. `http://www.utexas.edu/learn/html/spchar.html`, April 2001.

[22] F. Webber, J. Loyall, P. Pal, and R. Schantz. Building Adaptive and Agile Applications Using Intrusion Detection and Response. `http://www.dist-systems.bbn.com/papers/2000/NDSS/ndss00.ps`, February 2000.

[23] F. Webber, H.V. Ramasamy, J. Gossett, J. Loyall, J. Lyons, M. Atighetchi, M. Cukier, P. Pal, P. Pandey, R. Schantz, R. Watro, and W.H. Sanders. Intrusion Tolerance Approaches in ITUA. `http://www.crhc.uiuc.edu/PERFORM/Papers/USAN_papers/01CUK01.pdf`, July 2001.

[24] Seth E. Webster. The Development and Analysis of Intrusion Detection Algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1998.

[25] Simson L. Garfinkel with Gene Spafford. *Web Security and Commerce.* O'Reilly and Associates, Inc., Cambridge, MA, first edition, June 1997.