EVALUATING SECURITY MEASURES OF A LAYERED SYSTEM

By

Sanaz Hafezian Razavi

Software Engineering, Islamic Azad University, Iran, 2003

A Thesis

Presented to Ryerson University

In partial fulfillment of the

Requirement for the degree of

Master of Applied Science

In the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2009

Author's Declaration

I hereby declare that I am the sole author of this thesis project.

I authorize Ryerson University to lend this project to other institutions or individuals for th purpose of scholarly research.

Signature

I further authorize Ryerson University to reproduce this thesis by photocopying or by othe means, in total or in part, at the request of other institutions or individuals for the purpose o scholarly research.

Signature

Instructions on Borrowers

Ryerson University requires the signature of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

EVALUATING SECURITY MEASURES OF A LAYERED SYSTEM

©Sanaz Hafezian Razavi, 2009

Master of Applied Science
Electrical and Computer Engineering
Ryerson University

Most distributed systems that we use in our daily lives have layered architecture since such architectures allow separation of processing between multiple processes in different layers thereby reducing the complexity of the system. Unauthorized control over such systems can have potentially serious consequences ranging from huge monetary loss to even loss of human life. Hence considerable research attention is being given towards building tools and techniques for quantitative modeling and evaluation of security properties. This thesis proposes a high-level stochastic model to estimate security of a layered system. It discusses evaluation of availability and integrity as two major security properties of a three-layered Architecture consisting of Client, Web-server, and Database. Using *Mobius* software, this study models the change in vulnerability of a layer owing to an intrusion in another layer. Furthermore, it analyzes the impact on the security of the upper layers due to an intruded lower layer. While maintaining a system availability of 97.73%, this study indicates that increasing the system host attack rate in the Database layer from 10 to 20 will reduce system availability to 97.55%. Similar modification

made to a Web-server layer will contribute to 97.04% availability. This set of results imply that increasing attack rate in Web Server layer has a more severe impact on system availability, while the same modification in Database layer will less severely influence system availability.

Similar results have been gathered when measuring integrity of the system under identical set of modification. At system integrity of 96.88%, increasing host attack rate in Database layer has resulted in achieving integrity of 96.68%; similar experiment for Web server layer resulted in system integrity of 96.57%.

Acknowledgments

Thereby, I would like to convey my special appreciations to my supervisor, Dr. Olivia Das, who has supported, guided, and supervised me through completion of this study thesis and for reviewing this document and offering valuable comments. I would like to thank my friends and fellow colleagues for sharing their thoughts and making the DAS lab a pleasant learning and working environment at Ryerson University. I am thankful to Bruce Derwin, Engineering support at Ryerson University, for his technical support and for allowing continuous availability of tools and technology. I would like to thank Ken Keefe and Shravan Gaonkar – Group members of Mobius team- for sharing their expertise, and providing me with assistance, and valuable comments on this study. The contributions of Sankalp Singh from the University of Illinois, is remembered and appreciated. Finally yet importantly, I would like to thank my lovely husband, Farhad, for being with me, patiently understanding, and assisting me every step of this thesis.

To my husband, Farhad, for his patience and support

Table of Contents

List of Figures

List of Abbreviations

ADSG	Attack-Defence Stochastic Game
ARG	Attack Response Graph
COTS	Commercial available Off-The-Shelf
DB	Database
DTMC	Discrete Time Markov Chain
ESPN	Extended Stochastic Petri Net
FTs	Fault trees
GSPN	Generalized Stochastic Petri Net
IDSW	Intrusion Detection Software
ITUA	Intrusion Tolerance by Unpredictable Adaptation
MRGP	Markov Regenerative Process
MRM	Markov Reward Model
MTTSF	mean-time-to-security-failure
PN	Petri-Nets
RBD	Reliability Block Diagram
SAN	Stochastic Activity Network
SGN	Stochastic Game Net
SMP	Semi-Markov Process
SPN	Stochastic Petri Net
SRN	Stochastic Reward Net
WS	Web server

CHAPTER 1

INTRODUCTION

1.1 Introduction and motivsation

Security is an essential property of a system structure that ensures protection of information and property from theft, corruption, or natural disaster, while allowing the information and property to remain accessible and productive to its intended users. It is important for hospitals to develop a comprehensive security management program to effectively support and maintain physical protection for patients, staff, and visitors. Available and accurate performance of almost all civic services, such as public transit operations, traffic management, intelligent transportation, airport management, and identity protection, and furthermore, confidential and protected state of any country's military goods and devices are just few instances where security is a vital key player.

The increasing dependency of almost all areas of science to computer systems has caused serious competition among systems with the same output but different capabilities. One of the demanding capabilities of any information system is to carry out its deliberated purpose while maintaining vital security properties even when the attack has occurred in the system. As has been discussed by Sanders et al. [1] and Madan et al. [2] there are three generations of software systems (Also known as *secure systems*): the first generation prevents any occurrence of security intrusion. The second generation emphasizes on detection of such security intrusions and alerts the system administrators. However, there were a number of factors that made these traditional approaches to the development of *secure system* less desirable. Development and validation of security systems were quite expensive, especially with the emergence of unbounded systems [4] such as the internet and large network infrastructures. Modifying a *secure system* was a difficult task, since any modification in a *secure system* required a costly re-validation. Furthermore, there was no second line of defence against attacks when a *secure system* was identified to be insecure.

These limitations have lead to the development of the third generation of software systems. The most innovative generation aims to tolerate the presence of intrusion in the system, and to reconfigure the system after an intrusion attack. Much of the older researches on system security focused on the details of designing the first and second generations. Although all these efforts have been helping the improvement of the system security, with today's complexity of distributed software systems, it is an essence to draw our attention to system-level security rather than emphasizing on the details, which are very different from one system to another.

Pursued by many researchers in the area of security system evaluation, modelling approaches have proven to be effective in examining security features of a system. While determining the actual security parameters of a system is quite troublesome, incorporating modelling as part of an integrated experimental design of a system allows using all information in the data and facilitates generation of estimated parameters of interest. This process characterizes more fully the behaviour of a system by allowing the user to speculate about how the system functions in more detail. In addition, models are constructed to deal with security in the early stages of a software system development. The modeling process, furthermore, enables simulation of experiments before performing them, which in return leads to more cost effective experiments, designs, delivers, and production.

1.2 Research Objectives

Nowadays, most of software systems have been designed based on distributed or layered architecture (e.g. banking systems, telecommunication systems). In an intrusion tolerant layered system, the impacts of each layer's failure are felt through the lack of ability of its caller to attain the desired service. Therefore, failures are propagated by the layered dependencies.

In addition, in spite of the efforts to design secure systems, to the best of our knowledge, however, there is very less methodology than what is actually needed to evaluate security of a computer system from quantitative point of view. More specifically, there is no systematic research done to evaluate security level of a layered intrusion tolerance system. Since it is almost impossible to build a complete secure system, being able to quantitatively evaluate security of a system is a very effective step in achieving higher level of security.

This thesis has proposed a model for evaluating security properties of a three-tier layered system. The model has been analyzed to study the effect of intrusions in lower layer servers on the security of higher layers. The model can be easily extended to n-tier systems, however, that would require more computing resources. There are various models of failure like fail-stop, fail-silent system, and fail-safe failure. This thesis considers fail-safe, where the system is producing arbitrary output, but it is clearly junk.

The availability of important information on today's software systems and the increasing dependence on various distributed applications have led to an impartial increase in the variety and complexity of intrusions. An *intrusion* into a computer system is defined as any possible outcome of an attack, which causes the designed system not to behave in a way its designers expected [1]. For example intentional sending of an ill-formed message to an application, causing it to crash; intentional causing of a buffer overflow in an application, in turn causing the application to run arbitrary code with its own privilege; and exploitation of an operating system vulnerability to gain unauthorized system administrator privilege [5]. Besides damages caused by intrusion, there are damages that are not included under definition of "intrusion" – in this thesis, referred to as "random failure". A few examples of random failure are turning off the electricity power supply, and discovering a bug in the system during everyday computer system use [5].

1.3 Previous Studies

Due to the large variation in the nature of studies reviewed in this thesis, this section will be delivered with two separate contents: (1) Dependability related studies and (2) Software architecture related studies. Evaluation of system dependability is a well-known study criterion, which has been explored by the researchers. In contrast, an evaluation of security measures is a new interest. However, most of the dependability context can be adopted and introduced to security area as well. Software architecture is considered as an effective element in measuring software dependability attributes [44], [45], [46], which also can be extended into software security.

Dependability related studies

The concept of dependability is defined as a property of computer system such that reliance can justifiably be placed on the service it delivers. Dependability attributes include reliability, safety, maintainability, availability, integrity, and confidentiality. These concepts will be defined later in chapter 3.5. Combining integrity and availability with respect to authorized actions, together with confidentiality, leads to *security* [4]. However, there is a significant difference between dependability and security: while dependability analysis presumes that random events in the software or hardware cause the system failure, security analysis bases its assumption upon the failure caused by human intent. Thus, not only security failures are usually correlated, the attackers may also learn from the previous attacks.

Different models have been used in dependability studies with extensions into security applications. Each model concentrated on particular levels of abstraction and system features. The remainder of this section will review a few classes of model illustration methods, including Combinatorial, model checking, and state based stochastic methods. *Combinatorial models* include Reliability Block Diagram (RBD) [6],[8] ,[9] ,[10] , Fault trees (FTs) [11], [12], [13], [14], and Attack Trees [15][16], [16], [18]. These models do not specify all possible system states required to achieve a solution. The drawback to these models is that they do not easily capture certain characteristics such as stochastic dependence and imperfect fault coverage.

Model checking is another method suitable to investigate dependability and security of a system, which is based on a reachability analysis of the model state space [1]. Some states reflect harmful conditions. Therefore, the idea behind model checking method is to examine the state space implied by the system. This method explores the entire state space and locates those uncovered; this allows the method to find a path for state transition, leading to an uncovered state. Instances of application of this approach are described in [19] and [20] where model-checking algorithm was used in order to analyze computer programs for security flaws. Ritchey et al. [21] and Sheyner et al. [22] have used this approach to model attacks on systems. One important weakness of this method is the size of state space.

While *Combinatorial methods* are somewhat limited in expressing the stochastic behaviour of security systems, M*odel checking* methods are much more comprehensive and allow explicit modeling of difficult connections. *State based stochastic* methods have been extensively developed when compared to other versions of model checking algorithms earlier used in

mathematical modeling; furthermore, the more recent methods denote probabilistic conjectures of time durations and transition behaviours. Markov Reward Model (MRM) is an example of this method group which has been explored by several researchers [23][24], [25]. Referred to as *largeness*, it is a major disadvantage of system states with complex structures consisted of numerous constituents. This matter has earlier been investigated and two solutions of *largeness avoidance* [26], [27], [28] and *largeness tolerance* [29], [30] have been suggested. Other instances of state-based stochastic model have been described by Ciardo et al. [31], and Kulkarni et al. [32] where the former used Semi Markov Process (SMP) and the latter used Markov Regenerative Process (MRGP).

More recent studies have benefited from higher levels of model representation in order to model the dependability and security. For instance, Interactive Markov chain (IMC) [33], Stochastic process algebra [34][35], Petri-Nets (PN) [36], and different extensions of Petri Net like Stochastic Petri Nets (SPN) [36], [38], Generalized Stochastic Petri Net (GSPN) [39], and Stochastic Reward Net (SRN) [40], Extended Stochastic Petri Net (ESPN) [41] and Stochastic Activity Network (SAN) [42], [43]. This thesis has used the Stochastic Activity Network modeling approach. This method allows the automatic generation of large Markov chains, therefore, the richness and diversity of the whole system can be conceptually explained without worrying about low-level details. SAN method, furthermore pose the following features: (1) Graphical models, (2) Simplicity in learning and usage, (3) Mathematical foundations, where the models can be automatically checked and analyzed by software tools (i.e. Mobius), and (4) Flexibility, that is the possibility of defining general functions for input and output gates in programming languages.

Software Architecture

The software architecture determines how various components, comprising the software, cooperate with one other. It also defines the component's deployment mapping on the available hardware. The extensive use of Object Oriented and Web-based software systems has led to increase in applications of component based software development. In this approach, each component may be Commercially available Off-The-Shelf (COTS), developed either contractually or in house and not necessarily in the office. Therefore, different groups in various

environments may participate in software development. As a result, the black-boxed approach – modeling of the system's dependability or security as one unit, based on the overall failure process of the software [47] – will not be appropriate any longer.

Earlier studies on software architecture, have proposed different approaches to system dependability assessment, including state-based [49], [50], [51],[52], path-based [53], [54], and additive algorithms [55]. Among these approaches, state-based algorithms used composite models and then combined them into a single model. The resulting model was then solved to evaluate the measures of interest. State-based algorithms have been exercised in this thesis to assess system security. The assumed software system in this study is an interactive software system having a layered architecture, wherein the system responds to the users' inputs and each layer of the system interacts only with its first upper and lower layer.

1.4 Thesis Contributions

Exploring related literature has proven to us that no systematic study has ever been completed to evaluate security measure of a layered system. The unique characteristic of this study is the application of Stochastic Activity Network (SAN) models in capturing the impacts of attack occurring on various layers of a distributed layered system. In addition to that, constructing a hypothetical model allowing analysis of security properties of a layered system has been of more importance than developing an actual system. Furthermore, having a modular nature, the proposed SAN models can be adapted to model any other layered intrusion tolerant system.

This thesis has considered both types of failure: *intrusion-based* failure and *random failure*. Regarding the concept of intrusion, this study, however, has focused on intrusions that may cause a system to deliberately fail delivering service or to deteriorate performance to a point that has not been intended. Hence, the provided definition for intrusion concentrates only on denial-of-service intrusions. The intrusions that merely probe a system's vulnerabilities are beyond the scope of this research.

Today's software systems are made up of numerous small entities, globally distributed, which are executed when called, and act as parts of a complex system [48]. Thus, evaluating their security attributes requires substantial amount of information about the software architecture.

Moreover, analysis of the security measures in early stages of a software life cycle facilitate changes in the system design, hence, the final software system could provide services of higher quality. On the contrary, for an existing software system, it is yet necessary to predict the impact of any modification in different entities of the system on the overall system's security. Furthermore, desired security levels of the software system must be maintained, while allowing for changes in a particular entity, or adding and removing any component in the system.

The proposed model [84] in this thesis has evaluated security measures; similar to the study conducted by Sharma et al [52], the approach utilized in the proposed model capture the impacts of software architecture (i.e. layered architecture) on evaluating the desired measures. Sharma et al [52] study, however, focused on performance evaluation while the proposed model in this study has performed assessments of security attributes. This model is also identical to the study accomplished by Singh et al. [5]: both studies have benefited from the Stochastic Activity Networks method and the Mobius tool to evaluate security measures. Yet, the software architectures recommended in the proposed model demonstrate significant differences.

A new model has been developed to evaluate security attributes of a layered intrusion tolerant system, composed of three levels: Client layer, Web server layer, and Data base layer. A significant contribution of this study is related to the utilization of Stochastic Activity Networks (SANs) in the model solution technique to assess security attributes of a layered system. The proposed model is briefly outlined below:

- *Modeling a 3-tier system using SAN formalisms:* different SAN sub models for different entities of the system have been designed, which were then consolidated to form the entire system model. In each SAN sub models, marking value of each place represents the status of that particular component in terms of being in working mode, being intruded, or being failed.
    - *Host sub models*: two specific sub models for web server and database hosts have been designed to capture the behaviour of hosts and attacker against one another. The attacker behaviour and the learning process during the sets of attacks have been captured by the flexible attack rate in host sub models. The intrusion-tolerance nature of the software may contribute to variable marking values of the system for which the system remains operational even in presence of an attack.

- *Replica sub models*: similar to the host sub model, two different sub models for each replica of Web server and Database layers have been proposed, which represent the attacker behaviour and system response by changing the marking value of SAN's places.
- *Entire system model*: A composed model containing all the sub models in a fashion to denote the relationship between different entities of the models has been designed. This model has been used to compute the security measures.
- *Security attribute evaluation*: based on the composed model, security attributes of the proposed system under different condition have been determined and analyzed to capture the effect of different entities on the others.

- A software tool called *Mobius* has been used to create, solve, and analyse the model.
- *Model analysis:* Analysis was conducted for both host and replica entities in Web Server and Database layers. This analysis has accounted for (1) the Effects of changes in host attack rates on security measures, and the comparison of these effects, and (2) the Effects of changes in quality of Intrusion Detection Software (IDSW) on Security Measures.

1.5 Thesis outline

The remainder of this thesis is organized as follows:

- Chapter 2 will provide background information on which the foundation of this research is structured. It mainly explores dependability and its evaluation methodologies, layered systems, and security assessment techniques.
- Chapter 3 will describe the terminology used in the remainder of the thesis. This chapter will also provide detailed description of the proposed system architecture. A brief explanation of SAN models and Mobius tool are also covered in this chapter. Finally, the chapter will investigate all the assumptions for the proposed model.
- Chapter 4 will provide detailed information for all the sub-models designed in Mobius. This chapter will explore the structure of SAN sub models as well as their application to evaluate the desired security measures.
- Chapter 5 will provide a review of security measures desired for the proposed system. It will illustrate how presence of vulnerability in each component of the system would

affect the rest of the system; it further discusses the impacts on the security of the upper layers due to intrusion in the lower layer

- Chapter 6 will provide the reader with conclusions drawn from this study; it summarizes the work, and describes the achievements of the research. Furthermore, this chapter will provide recommendations on future study.

CHAPTER 2

RELATED WORK

Traditionally security validation has only been explored qualitatively (e.g. Security Evaluation Criteria [1], [57]). However, when quantitative approaches attempted, they were based on formal or informal methods. The former ones [58] tried to confirm that certain security properties hold given a specified set of assumptions, while the later methods [59] made use of expert teams to try to compromise a system. In contrast, dependability analysis has habitually used Stochastic Modeling. This approach has also recently been adopted and used in security analysis, in order to obtain quantitative system measures [1], [1], [60]. It should be noted that not all security attributes can be integrated naturally into dependability concept. The reason for that is in dependability context, the system failure's roots are fundamentally different from those in security violations. Dependability analysis supposes that random events in hardware or software cause failures while failure in security analysis are caused by human intent, resulting in security failures. However, still methods used in dependability area can be transferred into security.

This chapter introduces some of the previous works in security criteria along with their pros and cons.

2.1 Early work on State-Based techniques

One of the groundbreaking attempts in security area was a study by Littlewood et al. [61]. Two basic measures of security were defined in this study: mean effort to security failure, and mean time to security failure. In selecting these measures, time and effort were treated as two separate entities an attacker had to impact to cause a successful security breach. Alternately, the time required for a security failure to occur can be assumed a random function of effort, thus making the models more stochastic and complex. The authors considered the application of probabilistic methods (i.e. Markovian Model) to assess the security of a system. In their study, the similarities

between reliability and security were considered and were further discussed in a process exploring the "Operational security" measures. To do so, the authors characterized the attacker's behaviour against that of the system. The proposed Markov model in their study consisted of "working" and "security failed" states, which were specified as two states supporting the security measurement. In Littlewood et al. [61] study, only the very simplest type of problem statement was considered; similar to those for reliability issues concerning time to the next failure, and they have not well thought-out any of the other interesting parallels that could have existed with availability. The study concluded by not offering a solid solution to the defined issues; instead a set of new open questions were posed as closing remarks to this study that required further clarification when striving for a viable probabilistic security evaluation.

2.2 Probabilistic model based on attacker behaviour

In another study, Jonsson et al. [62] presented a probabilistic model to evaluate security of a system based on attacker's behaviour. In this study, security concepts were treated as two distinctive types: behavioural and preventive. Limitation in data availability forced this study to last up to two years in an attempt to gather all the necessary data regarding the typical attacker's behaviour. In the approach proposed by this study, attacker's behaviour was divided into three phases: one, the learning phase, two, the standard attack phase, and three, the innovative attack phase. Regardless of the cause, the probability to thrive an intrusion was small during the first and last phases, yet the probability of occurrence for a successful attack was considerably higher in the standard attack phase. Markovian model with exponential time distribution was employed in this approach. Considered as a significant progress towards quantitatively assessing the system security, this approach, however, only modeled the behaviour of an attacker that was an ambiguity source in security evaluation.

2.3 Privilege Graph

Ortalo et al. [63] suggested that an appropriate notion of state for a probabilistic system model would be the degree of privilege that an attacker has obtained. They offered a methodology for modeling UNIX security vulnerabilities known as a *privilege graph*. This graph corresponded to

a global model for the entire system. By incorporating the assumptions concerning an attacker's behaviour with the privilege graph, "attack state graph" was obtained. Furthermore, the authors described a technique for transforming the attack state graph into a Markov chain where the states of the resulting Markov chain have represented the enhanced privileges gained by an attacker as a result of a series of atomic attacks on the system. An interesting definition of reward used in this Markov is the "effort" needed to initiate a transition. By solving the obtained Markov model, several probabilistic security measures were assessed. The resulting Markov chain model was used for assessing the security attributes of a system by computing the mean time (or effort) required to send the system into a security failed state. To illustrate the application of their approach, an analysis was conducted using data obtained from measurements conducted on a large computer installation over a 21-month period. In spite of the interesting results obtained, the approach recommended by the authors had two limitations: (1), only the security of a particular system was assessed with respect to the known vulnerabilities, and (2), collection of a large amount of data was required to populate a constructed model. Therefore, such an approach was well suited only when discovering and assessing the impact of known vulnerabilities in an operating system. This approach proved less capability in predicting the relative efficacy of alternative intrusion tolerance techniques. In order to make such predictions, a higher-level approach was needed that could focus on the operation of the intrusion tolerance mechanisms, rather than on identified vulnerabilities.

2.4 Application of Semi-Markov Process to Model Attacker Behaviour and System Response

Madan et al. [1] made use of traditional stochastic modeling techniques to capture attacker behaviour and the system's response to attacks. This study illustrated behaviours of both the intruder and the security system in a single Markov model for an intrusion tolerant system called SITAR that has earlier been introduced. Based on particular attack scenarios, this study illustrated that there was a noticeable correlation between states and failure of availability, data integrity, and data confidentiality. Steady-state behaviour of the system was computed using this model, followed by determining the steady state availability of SITAR system. Because of the non-exponential behaviour of underlying stochastic model, this model needs to be formulated in

terms of a Semi-Markov Process (SMP). Based on the classification of the SMPs (i.e. *irreducible* SMP and SMP *with absorbing states*), two different types of security attributes have been pictured. The steady-state probabilities of various states were carried out from the irreducible SMP, which then lead to the computation of the steady-state availability. In contrast to the irreducible SMP model, the analysis of an absorbing state SMP model dealt with identifying the absorbing states, which were typically the security-failed states. This analysis was used to serve two main findings: one, to determine the time or the effort required for the SMP to reach an absorbing state in order to yield *the mean time to security failure* (MTTSF), and second, to learn about the probabilities of reaching different absorbing states. Utilizing these probabilities, the causes of different security violation types were identified independent of one another. Insufficient data, however, forced the study to incorporate unreal data in generating the results. This study was then concluded by introducing semi-automated and automated experiments to cover data shortage.

2.5 Partially Observable Markov Decision Processes

An alternative approach, introduced by Zhang et al. [64] is a state-based approach to monitor behaviour of attacker and defender in case of multi-stage cyber attacks on a typical network. The authors believed that attacker's behaviour played a key role in causing models of security evaluation to be complicated. They considered an attack in terms of its *effect* on the system rather than the attack itself. From the authors' perspective, this approach could facilitate categorizing different types of attacks within the same group, even if the attacks contributed to the same fault in the system. Another observation in this study was that attackers might change their behaviour with respect to the response received from the system. Authors integrated all of these notions in a simple state-space approach. They developed two objective-oriented models to assess system and attackers behaviours. Afterwards, the authors integrated and formulated the objectives and actions as a process known as Partially Observable Markov Decision Processes that also included the resultant impacts from the attacker and system and the underlying system states. In this process, the attacker and defender were represented as decision-making process and the process evolution was directed by the objectives. Furthermore, rewards were defined as the attacker and defender's objectives rather than underlying system states or their transitions.

The proposed model treated system states exclusively through observations produced by the key discrete states rather than continuous states, allowing for successfully limiting extensive growth in state space. Real data has been used to conduct the simulation-based experiments. Authors evaluated system security on the basis of action effects of attacker and defender, as well as the inferences of their optimal action policies. To develop their model, multi phases of system attack were considered; furthermore, the probability that a particular attack may occur under several phases with several sub-goals leading to the final purpose was also taken to account. Therefore, modeling attacker's behaviour allowed the system to detect attack instances prior to the final attack phase. Such modeling also enabled the system to form predictions about the attacker's next probable actions. In conclusion, investigation of better-objective-oriented models that provided more accurate quantification of attacker's intents, costs, and actions as well as the defender's cost-sensitive consequential actions in a complete multi-stage attack scenario was suggested to be postponed for consideration in future studies.

2.6 Attack-response Graph

A new approach of attack-response graph was taken by Madan et al. in[65].In their previous work, they used Markov chain model to achieve mean time to security failure, however creating the Markov model was not always an easy task. Therefore, they decided to define the Attack Response Graph (ARG) that illustrated the behaviour of attacker and system in the way that the arcs of the graph represented the attacker/system's action and the states represented the security-failed state. Going from start node of this graph to one of the other nodes, showed the path required to breach the system security. Transferring time between states of the graph could take random amount of time; consequently, the Stochastic Petri Net (SPN) driven from this ARG was a Generalized Stochastic Petri Net (GSPN). By creating reachability graph of GSPN, corresponding absorbing Markov chain was easy to obtain. This Markov has been used to compute the system security in terms of the mean-time-to-security-failure (MTTSF) measure for an intrusion tolerant system. The considered numerical instance confirms that the *MTTSF* could be increased by allowing the system to decide on more destructive measures when reacting to the attack. The difference between this approach and those of other researches was that in this approach the entire intrusion against the system was not accounted for. The only important

intrusions taken to consideration were those, which could be defended by the system. Moreover, this study only considered the security failure of the system and did not categorize this event as one of the more specialized causes. Such specialized causes for system security failure include failure of confidentiality, failure of availability, and failure of privacy. Such categorization of failed states could facilitate analyzing the Markov chain by computing another security measure based on the absorption probabilities of these states.

2.7 Game Strategies in Network Security

Another methodology used by researchers to assess system security attributes is known as the Game Theory. Lye et al. [66] presented a model to examine the network security and to determine the best-response strategies for the attackers and system administrators. In this model, the attacker and system behaviours were considered as two opponents in an absolute game of competition. This concept was illustrated using a concrete example in which an attacker invaded into the structure of a simple enterprise network that provided internet services such as web and FTP servers. In this example, the authors identified a set of specific states and made assumptions about state-transition probabilities. By using a non-linear program, the authors calculated the "Nash Equilibria" which provided the system administrator with ideas about attacker's behaviour in future states. This program also allowed the system to form logical decisions on selecting defence strategies. In addition, the study discussed the notion of utilities in order to combine the attackers' target, cost, and objectives. In spite of the benefits achieved through this method, a number of important disadvantages were pointed out. This study did not allow to manually enumerating the states for the attack scenario. Moreover, the game theoretical models consumed much longer time to speculate a multi-stage attack's strategy since they loosely treated the temporal correlation between attacker actions. The Game Theory model could hardly assist the developers to percept the attack schemes individually, and did not provide a framework to decide on the elements of a particular attack scheme. This is because in the Game Theory models, the attacker's objective is to achieve the highest payoff with respect to the system's defence component, and the strategies could only be inferred at the balanced state between attacker and defender. Additionally, high computational cost was identified as a disadvantage in utilizing the proposed method in practical applications.

2.8 Stochastic Game

Using stochastic techniques, Sallhammar et al. [67] modeled and evaluated the expected failure times of a computing system where the failure could occur on an arbitrarily or deliberately basis. The authors considered many effective factors on trustworthiness of the system including normal user behaviour, administrative activities, random failures, and attacks. Game Theory approach was taken to solve the behavioural model for system's future security and dependability. Utilizing this stochastic model, the states in which the system showed vulnerability to malicious faults were chosen as the game elements in a two-player, zero-sum, stochastic game. Game Theory model was based on a reward- and cost concept, making it possible to fine-tune the transition rates of the stochastic model for a particular threat assumption, such as the motivation and detection awareness of potential attackers. In solving the game, through weighting the transition rates according to probability distributions, the expected attacker behaviour could be revealed in the transitions between states in the system model. Finally, the corresponding stochastic process was used to compute operational measures of the system. The modelling approaches taken by these authors were similar to Madan et al. work [1]; except that decision probabilities were used when integrating attacker behaviour in the transition rates of the model. Moreover, this study illustrated modeling of unintended hardware and software failures, in conjunction with intrusions. In contrast to Lye et al. [66] study, Sallhammar et al. [67] modeled the game element outcomes generated by the attackers' actions regardless of them being detected by the system's Intrusion Detection Software (IDSW) mechanisms. Unlike the study performed by Liu et al. [67], this study used the same game model for different threat environments. Application feasibility study of time dependent success rates in computing more realistic strategies has been postponed for future research.

2.9 Markov Chain for Privilege

In another study, Jiang et al. [69] used Markov chain for privilege to model the attacker's behaviour. Again, the behaviour of attacker and system are viewed as two players of the game and an Attack-Defence Stochastic Game (ADSG) are formulated. Their work also assesses cost factors of cost-sensitive model and introduces the attack strategies prediction and optimal active

defence strategy decision. In the game theory strategy, gaining the greatest payoff for attacker and defender is their objective while it is considered that the system contains defence component as a whole and the strategies stay at the balance state between attacker and defender.

2.10 Stochastic Game Net

Wang et al. [70] applied Stochastic Game Net (SGN) to model and analyse the E-Commerce attack and defence. To construct the model, the authors defined the transition as possible actions that may be derived by the attacker or defender. Places represented the states of the system or the player according to the results of the actions. The reward gained by the player when an action is complete has been regarded as Reward element in the Game Theory model. By computing the strategy π as the choice probability under Nash Equilibrium and analyzing time and probabilities of different attack actions for e-commerce, the authors determined the optimal defence strategy and instructed it within the system administrator to use the obtained results enhancing the security of the system. This study was concluded by performing experiments showing that if the system was shifted towards a steady state, the successful attack probability was still independent of the attack rate. This result proved that when defining the defence strategies, the system administrators could consider the fixed attack probability and time for a given network. Hence, attack probability and intrusion period solely depend on the attacker's ability and the predicted reward, yet being independent of the attack rate. The proposed model in this study inherited the efficient and flexible modeling approach of Stochastic Petri Nets (SPN), while it also made well use of the game-theoretical framework from Stochastic Game theory. The authors believed that based on the obtained results, some effective defending mechanism can be designed.

2.11 Architecture-based Model

Sharma et al. [70] focused on an architecture-based unified hierarchical model for software reliability, performance, security, and cache behaviour prediction, which had not been widely explored in past studies. This study provided expressions for predicting the general system behaviour based on the individual components' features, which also accounted for second order

architectural effects for providing an accurate prediction. This approach also facilitated the identification of reliability, performance, security, and cache performance bottlenecks.

A hierarchical approach to forecast a variety of software system attributes based on the architecture and the attributes of the ingredient components were provided in this study. The novelty of this approach was in proposing a unique hierarchical model to assess reliability, performance, security, and cache behaviour prediction. By considering software architecture, this study made an original contribution in the field of cache-miss analysis. In this proposed absorbing Discrete Time Markov Chain (DTMC) model, the states of the DTMC at any time were given by the component in execution at that time. The arcs between states represented the transfer of control from one component or layer to another. The initial and final components receiving the control flow first and last denoted the initial and absorbing states of the DTMC, respectively. By assigning suitable rewards to this DTMC, reliability, security, performance and cache performances of the software system have been carried out. Two case studies were presented in this study to illustrate the practical application of the model. Furthermore, the authors provided the analytical techniques for sensitivity analysis of the system under consideration. Such analysis was necessary for systems in which the individual parameters were not accurately measured or estimated. Due to the hierarchical nature of the model, any changes in the system components' behaviour did not affect the model. A limitation of this approach was the complexity in modeling the concurrency of control flow using a DTMC and concurrently executing components that had to be modeled as a single state in the DTMC model.

2.12 Stochastic Activity Networks (1)

In a model-based validation effort, Stevens et al. [72] computed a networked intrusion tolerant information system called *Joint Battlespace Infosphere* (JBI). The Model-based results were used to design the system and to determine if a given survivability requirement was satisfied. In this study a hierarchical model was constructed based on stochastic activity network (SAN) using the Mobius tool [73], [74] to validate intrusion tolerant systems and to evaluate merits of various design choices. In this modeling, eighteen different SAN models for different system components were built and were combined using the Rep-Join composition formalism. The probabilistic model made use of an innovative attacker model. The attacker model had a

complicated and detailed depiction of different kinds of intrusions' effects on the system components behaviour (e.g. different failure modes). The attack model illustrated the process of discovery of vulnerabilities and further described their subsequent exploitation. Accounting for the connectivity of the components of the system, at the infrastructure and the logical levels, the model could call for an aggressive spread of attacks through the system. Furthermore, authors employed a probabilistic model to compare various design configurations, which allowed the system designers to select best configurations to maximize the intrusion tolerance provided by the system prior to system implementation. Stevens et al. derived this model based on experiments in which the survivability of the system was computed by measuring the probability of success for the transactions between the clients and the core. Finally, the model was used to illustrate that the system could meet a set of quantitative survivability requirements. To assemble such a model, a thorough specification of the system structure was needed.

2.13 Stochastic Activity Networks (2)

Singh et al. [5] conducted a research study using Stochastic Activity Networks (SAN) to validate and to evaluate the intrusion tolerant systems. The system modeled was part of the Intrusion Tolerance by Unpredictable Adaptation (ITUA) architecture, which intended to provide a middleware-based intrusion tolerance solution. The emphasis of this research was to study impacts of intrusions on the system behaviour and to explore the ability of the intrusion-tolerant mechanisms to handle those impacts. Assumptions were made regarding the discovery and the exploitation of vulnerabilities by the attackers to achieve such intrusions. The proposed model included different sub models for attacker, the intrusion-tolerance mechanism, the application, and the resource/privilege state of the system. Once a separate sub models for replica, host and management components (i.e. the ITUA components) were created, a complete model of the system was composed using replicate and join operations in a unique model. Moreover, the attacker model represented details of the intrusion itself as well as the effects of the intrusion. The authors constructed their attacker model based on the experiments conducted by Jonsson et al. [62] that suggested three distinct classes of attacks: script-based attacks, more exploratory attacks, and totally innovative attacks. Singh et al. also presumed the possibility of learning from successful intrusions for an attacker. Therefore, in the proposed model the corruption of a host in

a security domain enlarged the vulnerability of other hosts in the domain, since other host probably could have similar operating system versions and service configurations. In order to evaluate the interested security measures (i.e. unavailability and unreliability), the service was defined by an application to be improper when suffering a Byzantine fault (a third or more of the currently active replicas are corrupt). The authors gave insights into the relative merits of a variety of design choices by studying the variations in the mentioned measures in response to changes in system parameters. Furthermore, the authors conducted studies to evaluate the system sensitivity to variation of different parameters such as different distributions of a constant number of hosts into domains, Impact of Different Numbers of Hosts Distributed into a Constant Number of Domains and Comparison of Domain exclusion and Host exclusion Management Algorithms.

This thesis has illustrated the application of Stochastic Activity Networks for addressing the challenges involved in assessing system security measures, specifically assessing security measures of a three-layered intrusion tolerant system. Furthermore, the modular design of the model increases the compatibility to apply to any other 3-Tier software architecture. Application of SAN models will also allow defining new functions, when there is a desire to incorporate more security measures into the proposed designs in this thesis.

CHAPTER 3

SOFTWARE ARCHITECTURE AND MODEL ASSUMPTIONS

This chapter will provide fundamental concepts, which explore our proposed model. It briefly describes the structure of Stochastic Activity Networks and the Mobius platform, which have been utilized in this thesis as the modeling tool. Furthermore, the chapter will elaborate on the layered architecture used in the model and will outline model assumptions considered in the modelling process.

3.1 Stochastic Activity Network Modeling

Stochastic Activity Networks (SAN) are probabilistic extensions of activity networks, which allow building dependability, performance, and security models. Type of the extension is similar to the extension that constructs stochastic Petri nets from classical nets. Investigation of stochastic activity networks in detail is beyond the intentions of this thesis. A comprehensive description can be found in Sanders et al. [43] work. This section attempts to explain briefly the fundamental concepts of SAN model in order to provide a better understanding of the designed model.

SAN models are used to delineate a random process, which explains the system's behaviour. SAN model formalism benefits from four SAN primitives when specifying a model structure: Activities, Places, Input, and Output Gates. Each of these primitives has a graphical representation, which is useful for specifying a model in a concise and intuitive manner. Figure 1 illustrates the overall structure of SAN components.

- *Places* – represent the state of the modeled system. They contain natural numbers referred to as the *place's marking*. The model state is the ordered set of the place markings.

- *Activities* – characterize the actions of the modeled system. *Timed* and *instantaneous* are two types of activity. Their main characteristic is self-explanatory: the instantaneous activity does not take any time to be completed; while for timed activity, the required time to complete the activity is expressed as a random variable. It is also possible that an activity has different outcomes, which are specified as *cases*. Each case is assigned a probability, which can be a function of the model state. When an activity completes, an activity case is selected based on the case probabilities. Timed activities also have *activation predicates* and *reactivation predicates*. Note that these features have not been utilized in this thesis. An activity is referred to as *activated* when it becomes enabled.
- *Input Gates* – they determine if activities can become 'enabled' within a SAN structure. *Input gate predicate* and *input gate function* are two components of input gates. It is possible that an activity does not associate with an input gate. Otherwise, to allow an activity to be enabled, necessary requirements in input gate Predicate (i.e. *if* statement in Mobius platform) must be met. The *input gate function* specifies a change of model state that is executed when its associated activity completes.
- *Output Gates* – these gates allow changing of the state of the modeled system, once an activity is 'complete'. Output gates can also be associated with individual activity cases. Therefore, for activities with more than a case, Output gates may be used to specify the outcomes of every case. In addition, Output gates may be linked to an activity having no input gate to define the effect of the activity on the model's state.
- *Arcs* – are used to connect the places and activities. An arc stretching from a place to an activity symbolizes an implicit enabling condition. This implicit enabling condition requires the marking of the attached place to be greater than zero. A directed arc, initiated from an activity and ending at a place, specifies an implicit state change upon activity completion. This implicit state change results in increasing the marking of the place by one [80].

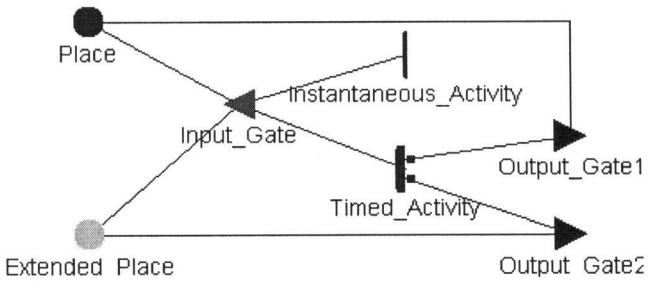


Figure 1 - SAN primitives

3.2 Mobius Workflow

A detailed description of the Mobius modeling tool is given by [74], [75].

- *Atomic Models* – the primary and most basic structure in Mobius is the atomic model, which can be parameterized with the use of global variables. It is possible to vary the values of these global variables and produce a set of experiments to evaluate the model for different parameter values.

- *Composed Models* – the hierarchical system modeling approach is supported by Mobius through a composed model. In this approach, atomic models are linked together through sharing state variables of different atomic models, allowing direct interaction among atomic models and the shared state variables. The relationship amongst atomic models is achieved by constructing a 'composed model'. Sub-model, Replicate, and Join blocks are components that are essential elements of a composed model. Mobius Replica/Join composition feature is illustrated in Figure 2. Sub-models are the building blocks required to construct larger models. A Join node establishes a number of state variables that are shared among its children. Replica node is a special kind of a Join node in which all children of a Replica are identical in sub-model type [75].

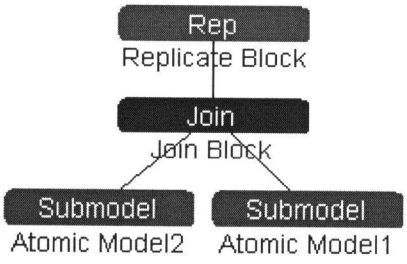


Figure 2 - Composed Models

- *Reward Models*: reward models are utilized to define measures of interest that the analyzer wishes to obtain from the system model. Once an atomic or composed model has been defined, a reward model can be created. When the model is simulated or numerically solved, the reward model identifies what data from the system needs to be collected.
- *Studies* – a study is created in order to indicate a set of experiments to be performed on a parameterized model in an automated manner.
- *State Space Generators and Solvers* – a numerical analysis of a Markovian model is performed in two steps: a state space exploration and the numerical calculation of a transient or steady state distribution, which is in turn used to evaluate rewards of interest.
- *Simulator* – to retrieve the generated results of a certain model, reward, and study, Mobius performs a stochastic discrete event simulation.

The dependency of above Mobius elements is illustrated in Figure 3.

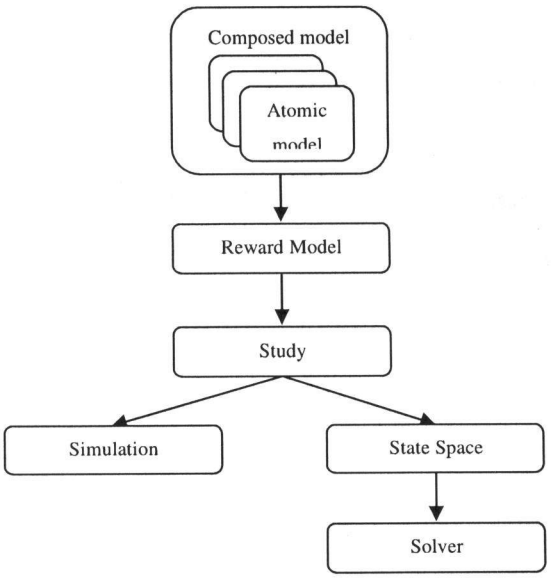


Figure 3 - Mobius Workflow

3.3 Overview of Layered Architecture

The objective of this thesis has been to model security attributes of a layered intrusion tolerant system. One of the most prevalent software architectures used in almost all client server systems is layered software architecture, which allows construction of software systems that can be decomposed into a few parts, where each part is at a particular level of abstraction with well-defined interface [78]. In a layered system, an entity at a lower level provides services to entities at a higher level. Multiple entities at a given level may use an entity at a lower level, but the lower level is not actually part of any of those. In contrast to layered system, traditional hierarchical systems were built on the is-a-part-of relationship, where an element at one level was composed of elements from the level immediately below it [52].

Layered architecture is utilized in almost all web-based systems where security is a very important factor. Therefore, security assessment of a layered system has become of high importance to system analyst and system architects.

In this thesis, an interactive three layered intrusion tolerant system has been considered. An interactive system responds to users' inputs. Three-tier architecture is typically taken up when a distributed client server design is needed, providing a significant improvement in system flexibility, reusability, and maintainability. In addition, this architecture simplifies the

complexity of the distributed processing system for the end user. Therefore, such architecture has been found well situated for internet applications, which are the most common applications these days.

Even though this work has focused on hypothetical applications of an assumed system, yet a common layered intrusion tolerant structure has been considered. By changing the value of assumed variables to the values of interest, the model can be used in any other system with the same architecture. Figure 4 visualizes three-tier client server architecture.

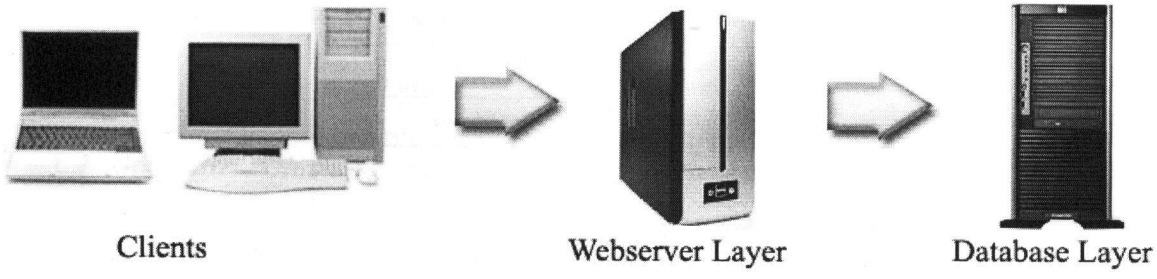


Figure 4 - Layered Architecture

This system allows users to interact with the first layer, which passes the request, if needed, to the second layer, which may then be passed to the next available layers. Once the last layer locates the information related to the request, it sends that information back, which then goes across the layers, until the user receives the output. Each layer works as a functional unit, which provides a defined set of services through the designed interface. Hence, it is not necessary that different components of a layer actually be placed in the same hardware. Nevertheless, various components of software from different layers might be located in identical or different machines. Once a layer is completed with its assigned duties, it either returns the generated results to the user or sends a request to its next layer in the same or different machine.

3.4 Software Architecture

The proposed system architecture in this thesis has taken the benefit of the replication method in order to tolerate break down caused by the presence of an intrusion or a random failure. It consists of client(s), middleware, Web-server (WS) layer, and Database (DB) layer where Web-server and Database layers consist of *Total_num_hosts* hosts. Figure 5 depicts the overall

structure of the proposed system. There are *Total_num_subs* sub-systems in the system; each includes one Web-server along with *Total_num_reps* replicas located in the upper layer and one Database host containing the same number of replicas in the lower one.

In the designed model, Client entity sends a request to the middleware. Middleware decides which sub-system to send the user's request to and if for any reason like sub-system's unavailability, middleware does not receive the desired response, it switches to other sub-systems for obtaining the desired service. When simulation process begins, each replica in a host receives a unique identifier. Replicas with the same identifier in two layers communicate with each other. It is assumed that at the beginning of simulation process all the hosts are in the running state. In addition, each replica would initiate or resume the running mode once its equivalent replica in the other layer starts running.

A sub-system would work when the following two conditions are met:

- Both of the sub-system's hosts are running (i.e. not failed)
- At least one set of replicas is operating.

The whole system will work until there is at least one sub-system available.

In case of any entity's failure (replica or host), its relevant party in the other layer is forced to stop running; it would resume once the corrupted entity is repaired. A hierarchical relationship among the system's components is presumed in which the system entity is parent of hosts and replicas are hosts' children. Whenever a parent fails, all its children will stop running until the parent's recovery allows all its children to restart. In addition to failure caused by human-based attack, the random failure for all entities is also taken into consideration. For failure of every entity of the model including hosts and replicas in both layers, an individual repair facility is considered.

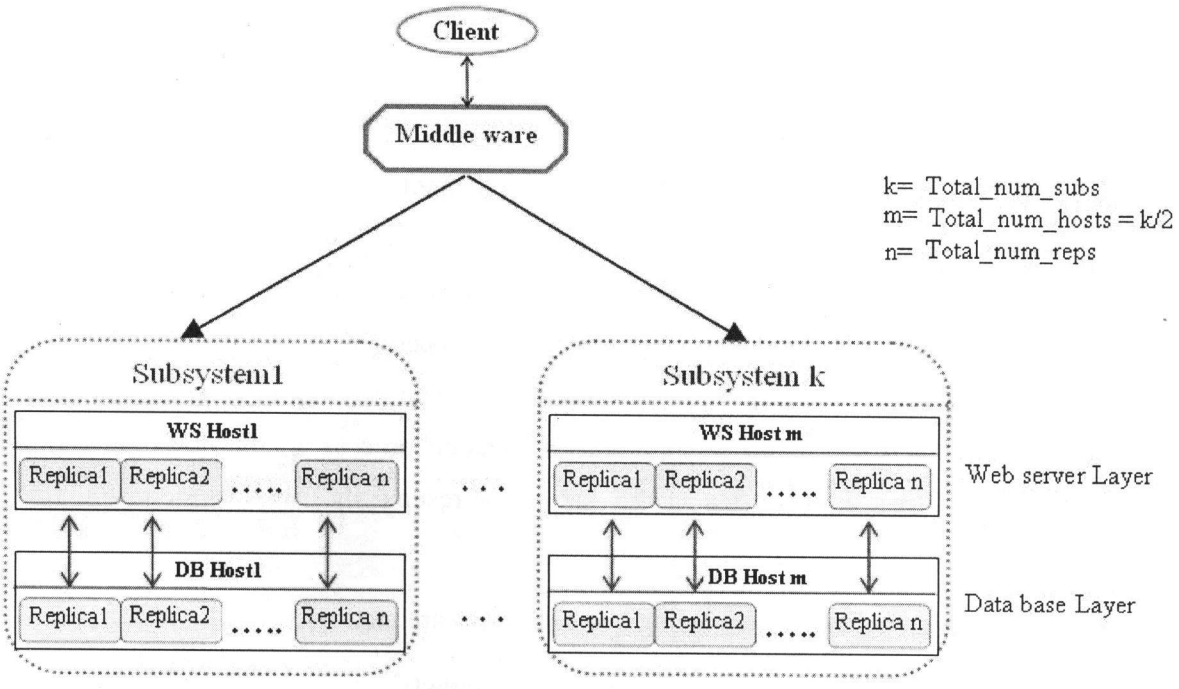


Figure 5 - System Architecture

3.5 Terminologies

For better delivery of material included in this document, the following definitions have been provided:

- *Host* – a computer or any other computing resource with an operating system and network interface cards
- *System behaviour* – the abstract output of the object system [76]
- *Inputs* – all possible operations to or against the system, including normal operations performed by system's defined users and operations, which result from intentional attacks against the system
- *Fault* – the progress that causes a system to experience vulnerabilities, to be added to the system during its design time deliberately and maliciously in some instances
- *Failure* – occurrence of potential faults in the system during the operational time which may result in security breaches
- *Security breach* – any possible input causing the system to behave against its security requirement [61]

- *Attack* – a malicious interaction fault through which an attacker may attempt to deliberately breach security property of the system [1]; similar to the work proposed by Littlewood et al. [61], this thesis has also considered attack behaviour as the following classification:
    - Passive attack, such as listening to public traffic and analyzing it,
    - Prohibited use of the system, such as the insertion of trap doors by privileged system users,
    - Attacks upon personnel, as in bribery or blackmail, which does not need any computing knowledge at all, nor even any interaction with the computer system itself
- *Intrusion* – any possible outcome of an attack [72]; an intrusion happens when vulnerability in the target component is located by the attack. As a result, the target component may behave in the manner not originally defined. This change in target component behaviour may be simple, as in a case when the intrusion is "masked"; or it can be very severe like when it compromises the target so that the targeted object could be utilized to establish a new attack. The intrusion is *prevented* when the attack fails to locate an existing vulnerability in its target. In contrast, an intrusion is tolerated when its impacts do not lead to unsuccessful processing of a published request.
- *Vulnerability* – a security exposure in any component of a system, which may be defined according to a successful intrusion
- *Repeated attack* – an attack is repeated when it happens on a similar target with identical properties (i.e. same attack type, same source).
- *Attack propagation* – a successful attack is propagated if the intruded target is compromised, leading the target to become a new basis for further attacks. When this source has access to a similar target with the similar vulnerability, the original attack may be repeated. The source can also initiate a new attack, which then leads to seeking for a new vulnerability in another target.
- *Security* – The ability of a system to prevent unauthorized access to the system [61]; more precisely security consists of combination of the following concepts:

- *Confidentiality* – the probability of the system to prevent an unauthorized user from having access to protected data [64], [76],
- *Integrity* – the probability of the system to prevent an unauthorized user from changing protected data [64], [76].
- *Availability* – the probability of the system to be ready/available for use in an instance of time [76].

3.6 Model Assumptions

Chapter 4 of this thesis describes in detail the proposed model components including sub models such as web-server and Database Hosts and their belonging replicas. There is no limitation in the quantity of applications to be included in this proposed architecture. Middleware is responsible for replicating any number of applications (i.e. Replica entity) and distributes them across the hosts in both layers. In addition, when receiving the user's request, it is the middleware's responsibility to select host and replica to which the request must be sent. The sending action is performed in a similar fashion as when the middleware receives the response from the upper layer and forwards it back to the user. Therefore, given all these tasks are completed by the middleware, the algorithm in which middleware has based its decision-making will not be an important matter; since these selections and connections of the elements are not prone to intrusion attacks. This model has focused on attacks on the system entity and the hosts and replicas within both layers.

All entities in our model are susceptible to attacks, which may potentially result in security breaches. In each layer (i.e. Web server and Database), there may be any number of damaged entities, which may or not be detected by the Intrusion Detection Software (IDSW). There exists a threshold for the number of detected and undetected damaged entities, which can be tolerated before a sub-system, or the entire system loses its availability (in case of detected damaged entity) or integrity (in case of undetected damaged entity). Byzantine fault tolerance [82], [83] utilizing authenticated Byzantine agreement under a timed-asynchronous environment has been assumed in the structure of the model. As a general convention in this thesis, the term 'corrupt' refers to entities whose IDSW has detected the presence of intrusion, causing full disability until

the repair process has completed. In contrast, the term 'damaged' refers to entities, which suffer from presence of intrusion but not yet detected by IDSW.

- *Applying Byzantine agreement to find system availability:* Less than a third of the total number of hosts in each layer can be corrupt, yet allow the entire system to stay available. In a similar notion, while less than one third of total number of replicas in each host is corrupt, the host with those replicas running on remains available. It is noticed that the assumed architecture is dynamic which means one entity may have been killed upon detection of corruption by IDSW, and a new one may have initiated to replace the corrupt one. When an entity (i.e. host or replica) receives a repair, it will remain on call until it is directed by middleware (in case of hosts) or host (in case of replica) to be replaced by one of the corrupt hosts or replicas.
- *Applying Byzantine agreement to find system integrity:* in case of integrity, the same agreement is applied for loosing sub-system or entire system integrity, except the term 'corrupt' is replaced by 'damaged'. Therefore, when an entity is under attack and remains undetected, with no interference with its performance, its outputs will not be trustable, leading to a loss in its integrity. However, the entire system's integrity will not be affected until more than one-third of hosts lose their integrity.

The IDSW discovers attacks on the host operating system, its services, and the replicas running on the host, although it is not guaranteed that all intrusions will be detected. Furthermore, it is probable that the IDSW has generated false alarms when there has not been any actual intrusion.

Through locating an attack in host or replica, their IDSW generates alarm, which causes the entity to stop running and the repair facility in the corrupt entity may be called by middleware. During the time when an entity spends in its repair facility, the entity party will also stop running until full recovery of the entity. Once the hosts' IDSW have located an intrusion in the host operating system or in the host service, the host with its entire replicas would be excluded from the system. Once the host is fully repaired, it will restart all its replicas. Therefore, if there were any corrupt replica, which has not received a repair service during the host repair, the corrupt replica will also restart to allow operation in a safe mode. The proposed model has assumed that the system is left on its own with minimum human intervention; thus, if excluding a host results

in running out of hosts, the system availability become zero until at least one host comes back to the system. The same assumption also applies for replica's exclusion.

The main objective of this thesis study has been to perform an assessment of security measures and to explore the effect on other entities due to an attack on certain entities, hence a secure mechanism for starting hosts, replicas and assigning the user request to the entities have been assumed. In addition, it has been assumed that the same number of hosts in each layer is running and each host runs the same number of replicas as others. Moreover, to keep the model simple a one to one relationship has been assumed between the hosts of two layers, as well as between the replicas in each host.

3.6.1 Attack Types

Detailed discussion on how security system boundaries have been defined in general is beyond the scope of this thesis, yet attempts have been made to propose a model to apply practical security problems on a wide basis. This thesis is focused on security assessment of a layered system. Although based on Jonsson et al. [62]'s experiments there are three different types of attack – script-based, exploratory, and innovative attacks – the proposed structure does not model attacker's behaviour in terms of attack type. From the security measure's point of view, the effect of these types is to lose trustworthiness and availability of the system. In this study, an attack may happen in different entities of the system including the hosts and the replicas within different layers. In the event of an attack in the host entity, host's operating system or any hardware or software residing in that host may become a target. This model is based upon the use of the updated commercial Intrusion Detection Softwares (IDSW), which are equipped with the latest information about a variety of attack types. Therefore, when assigning values to the variables regarding the quality of IDSW, it was assumed that the software was capable of detecting a majority of attacks.

3.6.2 Attack Propagation

An attack may occur in different entities of the system including the hosts and the replicas within the same or different layers. In the event of an attack in the host entity, host's operating system or any hardware or software residing in that host may be a target. Under the proposed algorithm, the targeted component can become a new source to launch new attack against the other attached

components. Thus, when a host is intruded, all the host's running replicas will be more vulnerable than the other replicas in the system. In other words, the presence of attack in a host greatly increases the probability of successful intrusion into all its replicas, which are running. This behaviour is modeled by assigning flexible rates to the replicas' attack activity. Therefore, replicas running on various hosts have different attack rates based on their host status. Detailed description of this assignment is given in section 4.4

One other type of attack propagation, considered in this thesis, is the spread of attack when a replica in upper layer has sent a request to the replica in the lower layer. A successful attack in each layer's replicas of the system may exploit through the next layer's replicas while two layers are communicating. If the sender replica has been attacked, it increases the chance of a successful intrusion in the receiver. Therefore, the system vulnerability does not always remain in the same stage and it would increase in the layer of which its above layer has been intruded. Thus, rate of attack in the lower layer is affected by marking its host's attack place as well as by marking its equivalent replica's attack place in the upper layer.

3.6.3 Intrusion Effects

It has been stated that once a host in a layer has been intruded and that the host's IDSW has detected the existence of an intrusion, the repair facility for this host will be called. Therefore, while this host is receiving repair service, all its belonging replicas will stop running.

Furthermore, when a failure occurs in the lower layer, this layer will not be able to return any results to the upper layer(s), resulting in unavailability of the upper layer(s). This implies that a successful attack in the upper layer host/replica will increase the vulnerability of the relevant replica in the lower layer, whereas a successful attack in a replica in the lower layer is a reason for unavailability of the relevant replica in the upper layer.

CHAPTER 4

SAN MODEL DESCRIPTION

A detailed explanation of the proposed model is provided in this chapter. Mobius software has been utilized to model the proposed model structure. In the last chapter, section 3.1 has provided a description of the four primitive elements of SAN models (i.e. Place, Activity, Input gates, and Output gates) and section 3.2 has explained the Mobius workflow and mentions about the primary role of Atomic sub models and Composed model in the entire model. The proposed model includes five atomic models, which are connected through a composed model. To achieve this connection, a few places in different atomic models are shared among two or more atomic models. The shared places have been identified using an asterisk (*).

In addition to shared places, there is another useful element called 'extended place', which allows the model to handle the representation of structures and arrays of primitive data types [75].

Mobius uses a C code for both input and output gates' functions. These codes, as also used in this model process, can be retrieved from Appendix A. In addition to these codes, all variables used in the model along with their values have also been provided in Appendix A. To provide a better understanding of the model description, a few variables or structures used in the model are defined below:

- Three variables 'Total_num_subs', 'Total_num_hosts', and 'Total_num_reps' have been used in atomic models as well as the composed model to represent the number of sub-systems (i.e. the combination of a host in each layer along with their replicas), hosts, and replicas in each host, respectively.

- Two user-defined variable types were introduced: 'Reps' variable, which is an array of *Total_num_reps* integers used as the type of a few extended places which maintain information about replicas. Another defined type of variable is referred to as the 'Sub' variable, which is a structure, containing two integers: *WS* and *DB* integers. This structure

has used as type of a few places. The value of this structure may be {0} or {1}, which refers to the status of host in each layer. (i.e. WS for web server layer and DB for Database layer)

4.1 Composed Model

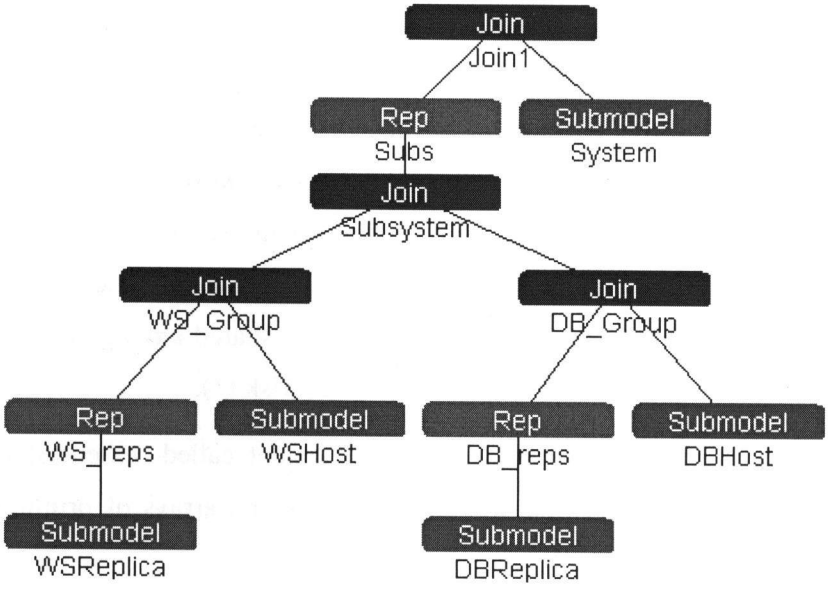


Figure 6 - Composed model

Five atomic sub-models, *DB_Replica*, *WS_Replica*, *DB_Host*, *WS_Host,* and *System* have been replicated and joined together to construct a complete model for the proposed layered system. Such model is illustrated in Figure 6.

As described earlier, there is a one-to-one relationship between replicas of two layers. Therefore, the same number of hosts and replicas are present in each layer. *DB_Replica* and *WS_Replica* sub-models are both replicated *Total_num_reps* times to create *DB_Reps* and *WS_Reps* respectively.

To attain complete DB and WS structures, each host sub-model must join its replica group. This concept is shown in the figure by joining the resultant node in previous step (i.e. *DB_Reps* and *WS_Reps*) to their relevant host.

A single sub-system is created once a host in the Web server layer is connected to a host in the Database layer.

A complete system architecture is achieved once the sub-systems have been replicated *Total_num_subs* times and the resultant node has joined the *System* entity.

4.2 System Sub-model

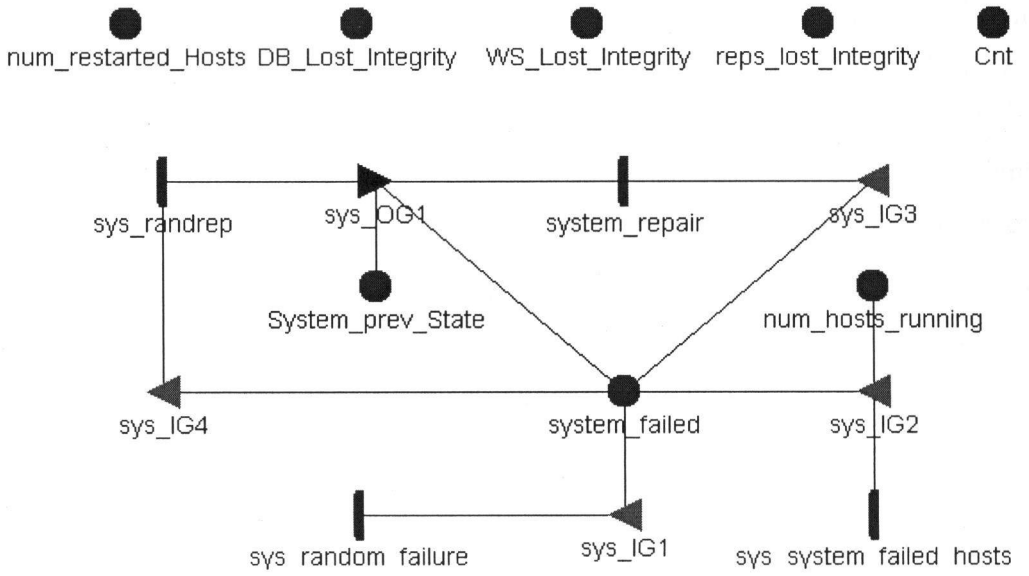


Figure 7 - System Sub model

A SAN view for the System entity is provided in Figure 7. This entity has been designed to capture the entire behaviour of the system. System failure occurs due to either a random failure or failure of the hosts running in the two layers. The activity, *sys_random*_failure, handles the random failure occurrence. Activity, *sys_system_failed_hosts*, fires when the place, *num_hosts_running**, has received a zero marking. This event happens once all hosts in the system have failed or stopped running. Each sub-system consists of two hosts, *WS_host* and *DB_host*; num*_hosts_running** initially has the value of *Total_num_hosts*, which is twice the value of *Total_num_subs.* Marking of this place is decremented whenever a host has failed and incremented upon host's repair. When any of the system failure activity fires, the place, *system_failed**, receives a marking of 2 in case of a random failure, and a marking of 1, in case of

failure due to the lack of running host. The marking of place, *system_failed**, illustrates whether the entire system is available (i.e. marking of 0).

All entities have been assigned two different types of repair facilities: one type for random failure and another type for intrusion-based failure. The strategy for assigning repair facilities to the entities is out of the scope of this thesis and it has been assumed that this mechanism has an acceptable performance. Depending on the value of the *system_failed** place, activities *sys_randrep* or *system_repair* may be fired, both causing similar modifications in the place's marking value. When the entire system is repaired, all the hosts will restart and reinitialize themselves. In addition, when the system fails because of the random failure, it is possible that there are still operating hosts, which need to be informed to stop running. To keep the hosts up to date regarding any changes occurring in the system status, the place *System_prev_State** will save the former status of the system prior its status.

The rest of places in the *System* atomic model that are shared among all the entities in the system will be described when their applications are discussed. These places are included in the *System* sub model to allow re-initialization of these places once the system has been repaired.

4.3 Host Sub-model

Figure 8 and Figure 9 illustrate the host SAN sub-models for a Database and Web server, respectively. Web server and Database host sub-models have almost identical activities and places; however, the rates of activities may vary. Rate of an activity is defined as the inversed mean time between two activity firings. Due to this similarity, for better presentation of material in this study, the prefixes DB and WS will be eliminated from the description names of activities and places. A host Sub-model (i.e. Database or Web server) may modify the host's running status through the activities: occurrence of an attack on the host, true or false detection of the intrusion, the occurrence of a random failure on the host, and finally the repair of the host. Among all the places within *host* sub-model, only *lost_Integrity,* and *host_failed* are used in security measurements. Fundamentally, both Database and web server *host* sub models in every subsystem are in a running mode, that is an indication of the sub-system being available (i.e. initial marking of *DB_host_running/ WS_host_running* is 1). There is a probability of an attack when a host is in running mode. *Attack_host* activity fires under the following conditions: (a) the host has not been under an attack, (b) the host is in running mode, and thus the entire system and the subsystem in which this host is running, are all available. Shared place, *Lost_Integrity**, is reserved for future utilization of measuring the security values. When a host is intruded, its retrieved data will not be reliable, as this data may have been modified by the attacker. Therefore, in spite of having the host's data available, the results generated by the host will not be reliable, while the host is under an attack that has not been detected by the intrusion detection software. Incrementing marking of the place, *Lost_Integrity**, is an indication of this concept.

A high quality, updated intrusion detection software facilitates the discovery of intrusion existence in the system; however, even well developed detection software systems have the potential for issuing false alarms. Yet, when an alarm has been accurately generated, there are a few places where new markings will be required. Two cases of the activity, *valid_alert*, represent false or true detection of intrusion; however, true detection has a higher weight. Upon true firing of activity, *valid_alert*, the host fails and therefore it will need to stop running. At this point, the host will not be capable of returning any feedback to its caller (either to the upper layer or to the middle ware), thus the data will not be available, but the integrity would not be lost anymore (decrementing marking of *Lost_Integrity**). Furthermore, number of running hosts will

decrement and the sub model will not be available anymore. The justification behind this failure is that the two hosts in two layers exchange data between one another and failure in one causes the other not being able to continue processing, thus resulting in subsystem's unavailability. Moreover, all replicas running on corrupt host will be stopped. Changing host status to failure causes *Host_repair* activity to be enabled. Alike to *System* entity, a random failure may occur in a *host* sub-model that has not failed. Similar actions will be taken by firing of *random_failure* activity, except that marking of place *Lost_Integrity** will not change.

Thus, the place, *host_failed*, may present three different markings:

(1) A marking value of zero is granted if the host has not been attacked and is in running mode, or when it is attacked but the attack has not been detected.

(2) A marking value of two is granted when a random failure occurs, and this triggers the *Host_randrep* Activity.

(3) A marking value of one is granted, after successfully detecting of an attack by IDSW, causing the *Host_repair* activity to be enabled.

The firing of activity, *Host_repair*, leads to the following adjustments:

(a) Re-initializing of the markings of places *host_failed*, *host_running*, and *host_attacked*

(b) Incrementing of the marking values of *num_hosts_running* place

(c) Decrementing of the marking values of place *Sub_Working*.

As explained earlier, when a parent is repaired, all its children will restart, forcing the marking value of the extended place, *reps_failed**, to become equal to its initial value, which is zero for all the elements.

It has already been stated that a single sub-system is available while both of its hosts (i.e. WS_Host and DB_Host) are running; that is no failure has occurred. When one of the hosts in a particular sub model fails, the *DB_Change_WSChange* activity in *DB_Host* sub-model or the *WS_change_DBChange* activity in *WS_host* sub-model will fire in order to bring the other host and all its replicas to a halt status. The extended shared place, *reps_prev_running**, has been used to maintain the last status for the replicas in the host – the host that need to be stopped due to failure in its other pair. Therefore, once the corrupt host has been repaired and is back to running mode, the replicas running on the safe host, which has been stopped, could return to their earlier status.

When the corrupt host has recovered and has started running, the activity, *DB_Change_WSChange/ WS_change_DBChange*, will fire once again to bring the correct host back to the running mode and its replicas back to their earlier status.

One other case in which the status of a host changes from running to not running is where the activity, *all_reps_failed*, has fired. If all the replicas running on a host have failed, although host has not failed, the host will not be able to respond to any requests. Therefore, the activity, *all_reps_failed*, will be enabled and fired. This behaviour justifies the sharing condition of the place, *reps_failed**, with a host and all the host's replicas.

As noted earlier, in certain instances the *System* entity enters the failure state, while the system hosts are still in running mode. Thus, the *system* entity will demand these hosts to stop operating. The activity, *Stop_sysFailure*, is responsible for transporting host and its belonging replicas to stop progressing when the system has failed. Thus, this activity ensures that while the *system* entity stays in the failure mode, no other activity in other entities will fire.

Once the system has been repaired, it will communicate with its entire hosts to reinitialize and to start operating again. The *Start_sysRepair* activity fires to restart the host and all its replicas. This activity is initiated only when the last status of the *System* entity and its status are not identical. Upon firing of *Start_sysRepair* activity in each host, all its replicas will also restart and the marking value of the local place *Has_restarted* will become 1 to ensure that this activity in the host will not fire again. Marking value of the globally shared place *num_hosts_running** also increases implying that a new host has been added to the system. *num_restarted_Hosts** and *Cnt** places are shared among the system entity, DB_Host, and WS_Host and are used to count the number of restarted hosts after the system has been repaired. It should be noted that no other activity in the entities would be fired until the *Start_sysRepair* activity has been fired in all hosts. During this time the marking value of the *System_prev_State** place stays 1.

The *ReInit_Prev* activity is fired once all the hosts in the system have restarted. The firing changes the previous status of the system as in the case with no failure (i.e. marking of *System_prev_State** equivalent to 0) allowing all activities in all the entities to fire if other

predicate conditions have been met. From this point on, the system will operate fresh from scratch, as in the case where no instance of attack had ever occurred.

Both host sub models (i.e. Data base host and Web server host) contain the activities earlier described; however, the rates of these activities are varied.

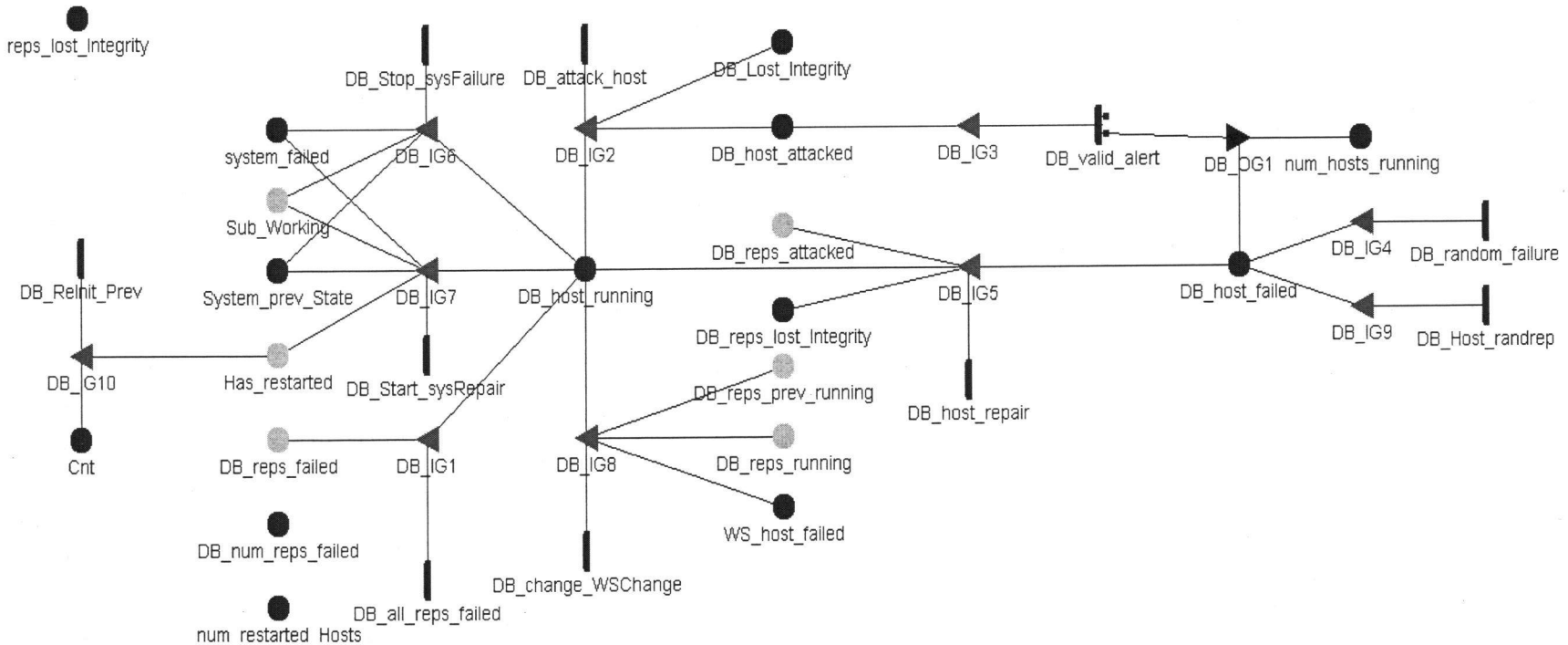


Figure 8 - DB_Host Sub-model

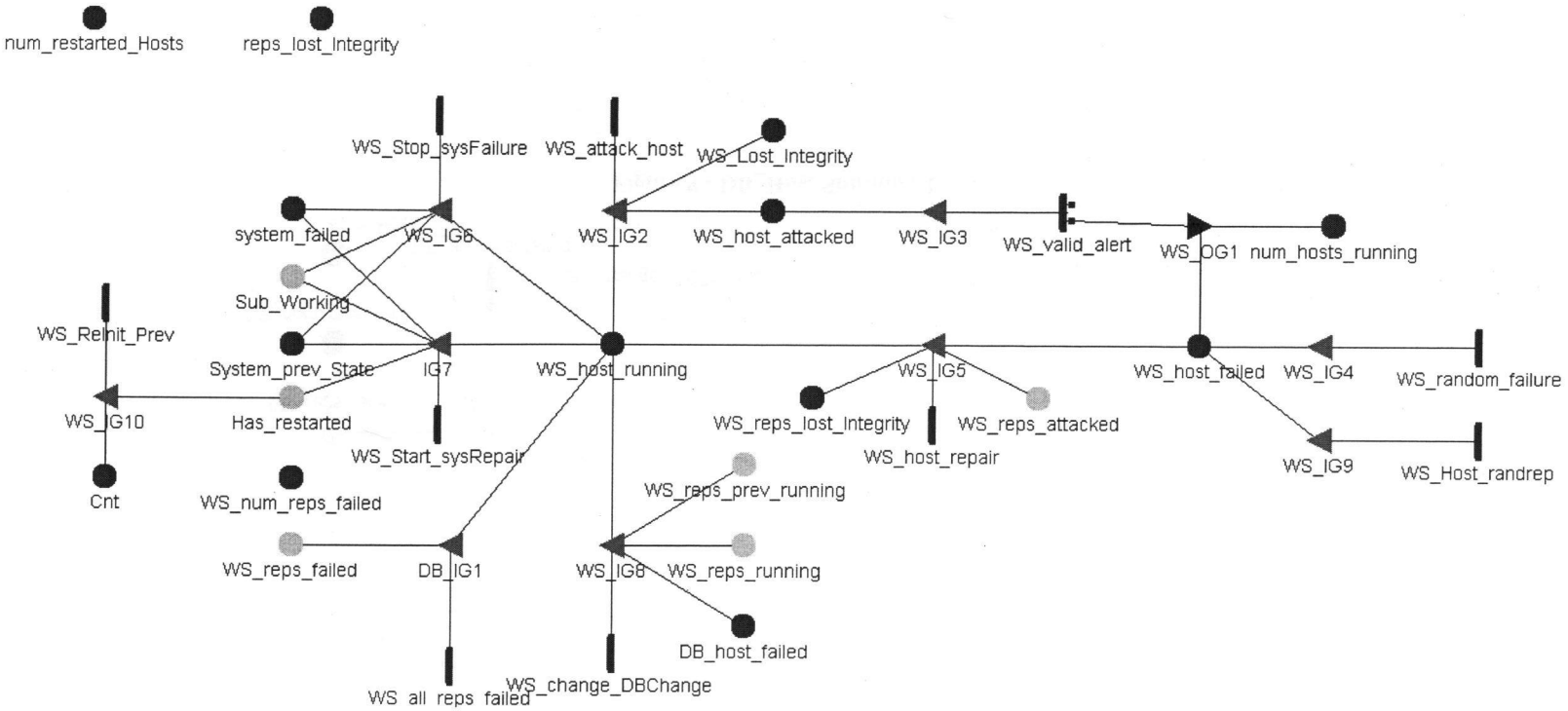


Figure 9 - WS_Host Sub-model

4.4 Replica Sub-model

Figure 10 and Figure 11 illustrate SAN models of *DB_Replica* and *WS_Replica*, respectively. Some of the shared conditions, which are required to permit firing of activities among replicas and host, are summarized below:

- All the activities and places in *DB_Replica* and *WS_Replica* are identical except the activities' rates.
- Similar to host entities, the common predicate conditions for all activities in the *Replica* sub model are: (a) both system and sub-system are working (b) system's previous status must be working (i.e. marking value of *System_prev_State* to be equivalent to zero).
- Another common precondition for all Replica's activities, except *Get_Repid,* is to have an identifier greater than zero.
- The entire extended place that has a "Reps" type, is an array of *Total_num_reps* elements, which is initially set to have a value of zero for each element. The i^{th} element of the array refers to the replica with identical code equal to *i-1*.

Since the concept of *lost_integrity* is also applicable to the replica entity, any replica in running mode may also undergo an intrusion event. Therefore, occurrence of an attack in a replica results in increasing the marking value of *reps_lost_Integrityis.* Thus, including more than one replica in each host will prevent sub-system integrity loss as long as the marking value of this place is less than one third of the total number of activated replicas in the sub-system according to Byzantine Agreement. A host will also stop running when all the running host replicas fail or stop running. Each replica in each host is distinguished from the others by a unique identifier, which is generated by firing of high rate activity *Get_Repid.* This activity fires only for replicas running on an available sub-system, which have not already received any identifier. Upon firing of *Get_Repid* activity, if the replica has not failed, belongs to a safe sub-system, and has not been in running mode, *start_rep* activity will be enabled. When *start_rep* activity fires, corresponding element of extended place *reps_running** will receive value of one, indicating that this replica is working.

Any replica in the running mode, which has not been attacked yet, is capable of being intruded. Upon firing of *attack_rep* activity, its corresponding element in extended place *reps_attacked** would get marking value of one to show the presence of intrusion in the replica.

Section 3.6.2 describes that under the proposed algorithm, when a host has been intruded; all the host's running replicas will be more vulnerable than the other replicas in the system. This behaviour has been modeled by assigning flexible rates to the replicas' attack activity. *Attack_rep* activity has a rate that is common between all replicas in each layer. When a host with running replicas is intruded, the multiplication of this rate by marking of *host_attack** allocates a higher rate to this activity. Therefore, replicas running on various hosts have different rates based on their host status. For example, the rate for the *WS_attack_rep* activity in the *WS_Replica* sub model is defined as:

WS_base_rep_attack_rate * (WS_host_attacked->Mark () +1.0)

If the corresponding host has not been intruded, place *WS_host_attacked* has the marking value of zero, which implies the above multiplication to be equal to WS_base_rep_attack_rate, while presence of intrusion in the host causes marking value for *WS_host_attacked* to be one. Therefore the activity's rate become 2* WS_base_rep_attack_rate. That indicates replicas running on an attacked host are two times more vulnerable than other replicas.

Another type of attack propagation, considered in this study is the spread of attack when a replica in upper layer (i.e. Web server) sends a request to the one in the lower layer (i.e. Database). If the sender replica has been attacked, it increases vulnerability of the receiver. Thus, rate of *attack_rep* in the lower layer will be affected by marking its host's attack place and also marking of its equivalent replica's attack place in the upper layer. Thus, the attack rate of *DB_attack_rep* activity in the *DB_Replica* sub model is defined as:

DB_base_rep_attack_rate * (*DB_host_attacked* ->Mark () +1.0) *

((WS_reps_attacked->Index (DB_rep_id->Mark ()-1) ->Mark ()) +1.0)

The above multiplication statement indicates that the rate of *DB_attack_rep* activity is affected by the marking value of the *DB_host_attacked* place and also by marking value of the relevant element in the *WS_reps_attacked* extended place. If any of these two places has marking values greater than zero, the rate of *DB_attack_rep* activity will become twice as it originally is (i.e. if

both *DB_host_attacked* place and *WS_reps_attacked* have marking values greater than zero, the rate of *DB_attack_rep* activity quadruples its base value.)

Moreover, referring to description of host sub-models, when an entity has been intruded, although it returns a feedback to its caller, the response will not be trustable due to a loss in the integrity. Therefore, occurrence of an attack in a replica results in increasing the marking value of the *reps_lost_Integrityis* place. However, since each host includes more than one replica, the integrity of the subsystem will not be lost until the marking value of this place has become more than one third of total number of activated replicas in the subsystem (Byzantine agreement).

Once *reps_attacked* place receives marking value of one, the *valid_alarm* activity will be enabled, which may produce a true or false alert, similar to the case described for host sub-model. Firing of *valid_alarm*'s true alert leads the *Replica* model to the failure mode, where the *rep_repair* activity will be ready to recover the replica. In terms of modification in places marking value of the model, firing of the *rep_random_failure* activity will also have the same consequence as that for *valid_alarm.*

In both cases replica will stop running and will designate a value of 1 in its equivalent element in the extended place, *reps_failed**. The only difference between the outputs generated by the firings of the *rep_random_failure* and *valid_alarm* activities is decrementing the marking of place *reps_lost_Integrityis* after firing of the *valid_alarm* activity.

When a replica has been repaired, it will remove its flag from extended shared places *reps_failed** and *reps_attacked** and will start running once again. As earlier mentioned, when a host status in one layer changes from running mode to stop running mode, and vice versa, the other host in the other layer has to follow the same status modification. Similar behaviour will exist for the replicas. Activities *DB_Changes_due_to_WSRep* in DB_Replica and *WS_Changes_due_to_DBRep* in WS_Replica sub models are incorporated to perform such adjustment. These activities will fire under two distinct conditions: (1) when the relevant replica in the other layer (e.g. Rep1) has failed- firing of this activity will bring the safe replica (e.g. Rep2) to the stop running mode; Rep2 will remains in halt mode until Rep1 has been repaired, (2) once the failed replica (i.e. Rep1) has been repaired, *Changes_due_to_DBRep* activity fires once again to transport the safe Replica (i.e. Rep2) from the stop running mode to the running mode.

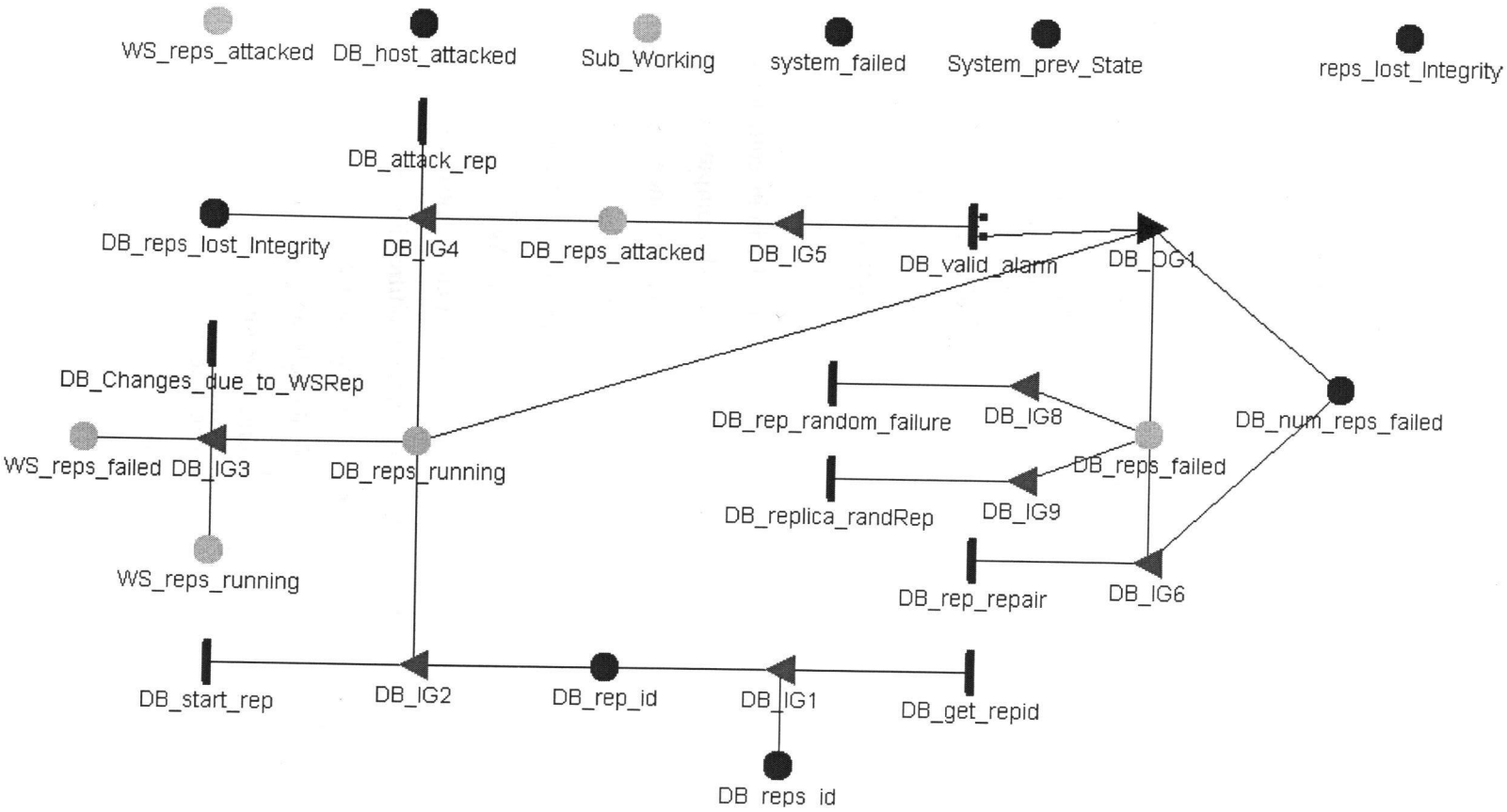


Figure 10 - DB_Replica Sub-model

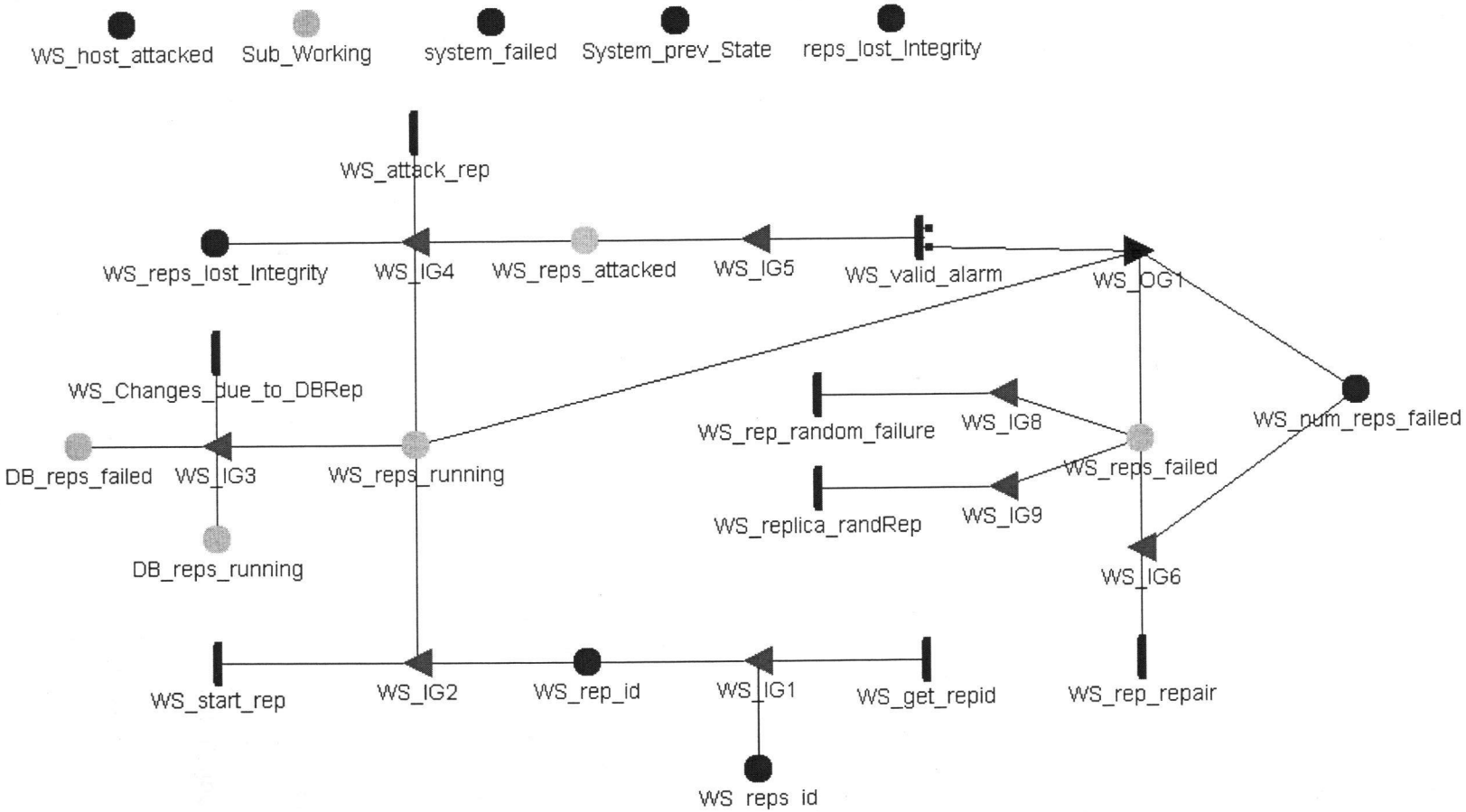


Figure 11 - WS_Replica Sub-model

CHAPTER 5

RESULTS

This study has proposed a model to assess availability and integrity as two key system security measures of a layered structure. Mobius tool has been used to design the stochastic activity network sub-models, and different studies were conducted to define the interested security attributes. The security measures evaluated by this model for specified time interval include:

- *System availability*: unavailability of a system represents the fraction of time the system does not respond due to the following reasons:
    - All the sub-system's hosts have failed or stopped running. This occurs due to a random failure or failure after successful detection of intrusion in the host. The system is then available while at least one of its sub-systems is available.
    - Failure of all the replicas running on all the hosts while the hosts are in running status. This failure causes the host to stop responding and consequently do not answer the caller.
- *System integrity,* which indicates the portion of time the system, is available but the service is inappropriate in the study interval. The system middle ware receives request from one of the sub-systems and therefore the integrity of the system in any fraction of time is defined as the integrity of the sub-system, which returns the response to the middle ware. The inappropriate service of the system may be due to:
    - The presence of intrusion in at least one third of total number of hosts in the entire system (i.e. Byzantine Agreement for hosts).
    - The presence of intrusion in at least one third of total number of replicas in the entire system (i.e. Byzantine Agreement for replicas).

The availability and integrity security measures, discussed in these studies, are quantitatively computed in form of the reward functions in the Mobius software. In the rest of this chapter various studies we conducted by the model are explained.

In the first and most basic experiment, we evaluated the security measures for different number of sub-systems as well as different distribution of replicas in each sub-system's host. All the following experiments inherit the study parameters from the first one except the parameter of interest in that particular study to assess the effect of the specified parameter in the security attribute of the system.

Studies 5.2 and 5.3 compare the intrusion tolerance of the system when the rate (i.e. the number of time an attack against the host happens in each time unit) of attack against each layer's host increases. These studies has performed on the database layer and then on the web server layer. Finally, in Study 5.4 the security measures of two layers are compared together to show how each layer influences the other layer's security measures.

The next three studies (i.e. Studies 5.5, 5.6, and 5.7) compared security attributes of the layered system while the quality of intrusion detection software varies. As already stated, it is possible that intrusion detection software alarms inaccurately. By increasing the quality of IDSW, the number of false alarm reduces. Therefore, we studied the security measure of the system in case of increasing valid alarm rates of IDSW in each layer's host. Also as previously stated, the last study of this set shows how quality of each layer host's IDSW may cause the other layer to have higher or lower security measures.

In three more studies (i.e. studies 5.8, 5.9, and 5.10), each layer replica's attack rate rises, and the impact of this increment in each layer and also in comparison of both layers are evaluated.
The last set of studies (i.e. studies 5.11, 5.12, and 5.13) calculates security measures of the system when the quality of replica's IDSW in each layer changes. Some of these parameters (for example Attack_host_rate, Base_rep_attack_rate, valid_alert_rate and Rep_valid_alarm_rate) have been taken from Singh et al. [5].

For better consistency, duration of one hour has. been considered as a time unit. All the experiments have been simulated by Mobius for an interval of 1-100 time units (hour).The followings are the parameters, which have been used with the same values in all the studies:

- *Total_num_reps/ Total_num_subs*: For each experiment, except the first experiment, the number of replicas is set to be two and four. In addition, the number of sub-system is set to be two and four, which means four and eight hosts respectively.

- *Random_failure_rate/ Rep_random_failure_rate*: While very rare to occur, random failures such as electrical power outage are also taken to consideration. Random failure rate for Database and Web server host is 1.0E-4. Replica's random failure rate in each layer is equal to 0.001.
- *Host_randrep_rate/ Host_repair_rate*: A variety of repair facilities for attacked failures and random failures are also included in this model. Each layer's host's repair rate in case of random failure is set to 0.001 and in case of failure caused by attack is sat to 5.0.
- *Replica_randRep_rate/ Replica_repair_rate*: replicas repair rate for random failure equals to 0.01 and repair rate after failure by intrusion is 10.0.
- *prob_succ/ rep_prob_succ*: Intrusion detection probabilities are always considered to be 99% in case of existence of attack in host and to be 90% in case of presence of attack in replica.
- *Replica_getid_rate/ Start_rep_rate*: the rate of assigning an identifier to each replica in each host is 100.0 and the rate of starting each replica is equal to 50.0.
- *Change_WSChange_rate/ Changes_due_to_WSRep_rate*: all replicas and hosts are always monitoring the status of their part to switch their own status as needed from running to stop running or in opposite manner. The rate of this monitoring is 10.0 for both components.
- *All_reps_failed_rate*: Each host may switch to stop running status if all of replicas running on it have failed. The rate of this check to decide whether the switch should occur or not, is set to 5.0 for the hosts in both layers.
- *System_failed_hosts_rate/ system_repair*: the whole system fails when the entire hosts in both layers stop running. The rate for this event is 1.0. Repair rate for the whole system is also 1.0.
- Start_sysRepair: when system is repaired, all the hosts reinitialize themselves. The rate for each host re-initialization is 5.0.

- *Attack_host_rate/ Base_rep_attack_rate*: rate of attack against host is 10.0 unless it specified in the experiment. Base rate for replica attack is equal to 40.0, which may change in different replicas based on their host attack status.
- *Valid_alert_rate / Rep_valid_alarm_rate*: rate of true detection of attack for host and replica is set to 5.0 and 10.0 respectively.

In the following studies, all the presented results have a 90 percent confidence interval. For better illustration, in addition to the base graph of the experiment for a specific variable, two separate graphs are plotted for which smaller range of security measures (i.e. Availability and Integrity) are shown. For instance, figure 14.1(a) is divided into graphs 14.1(b) and 14.1(c) in which micro variations of availability measures for variable *DB_Attack_host_rate* are better presented.

5.1 The Effect of Replica Distribution and Sub-system Quantity on Security Measures

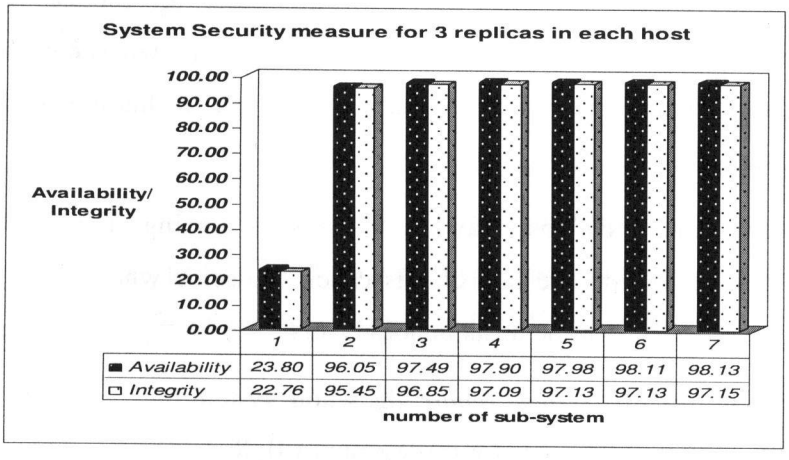


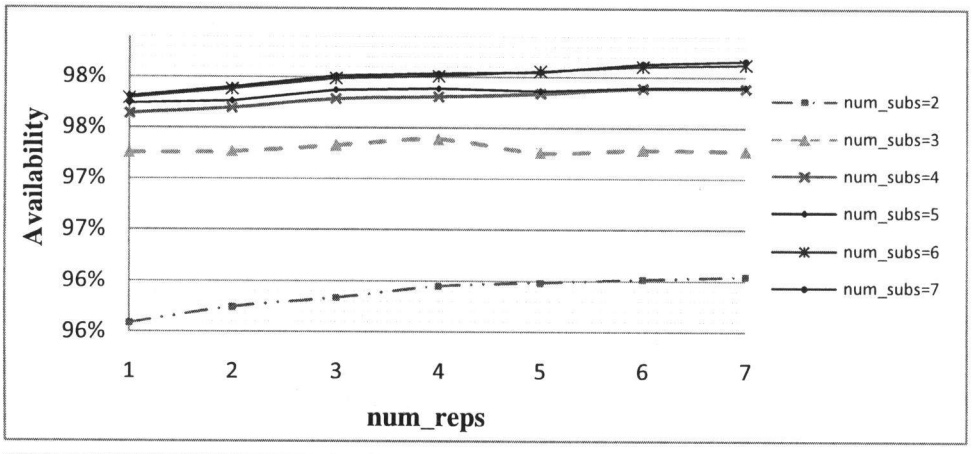
Figure 12- System security measures for different number of sub-systems and different distributions of replica in each host

Figure 12 contains graphic illustrations of the results obtained from the proposed model. This figure shows the changes in system's availability and integrity for 1 host in each layer, 3 replicas running in each host, and sub-systems varying from 1 to 7. As can be seen, there is a significant

difference between system security measures when one sub-system is considered compared to cases with more than one sub-system for the whole system. However, having more than four sub-systems does not change the security attributes very considerably. Therefore, considering the expected outcome as well as the target cost-to-benefit ratio, it is the system administrator's role to determine if more than four sub-systems must be included in the system.

Figure 13 presents the effect of increasing number of replicas in each host on system security measures when the number of sub-system is kept constant. Exploring the variations of availability and integrity in these graphs, it is clear that increasing the number of replica will increase system security attributes, accordingly. Once again, it must be noted that while levels of availability and integrity significantly alter when moving along the X-axis from 1 to 4 (i.e. number of replicas changing from 1 to 4); the security measures value will not radically change when we go further. Based on this observation, one could determine it adding more than four replicas in a host is not a beneficial use of available resources.

13.1



13.2

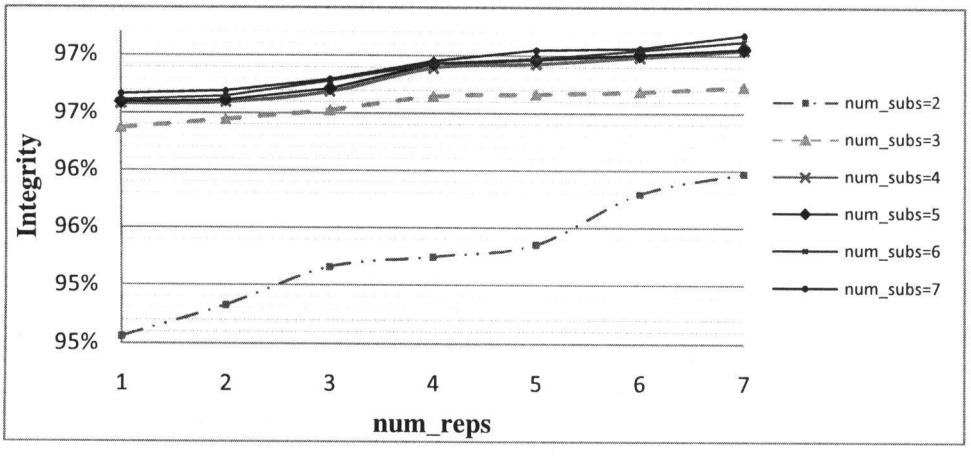


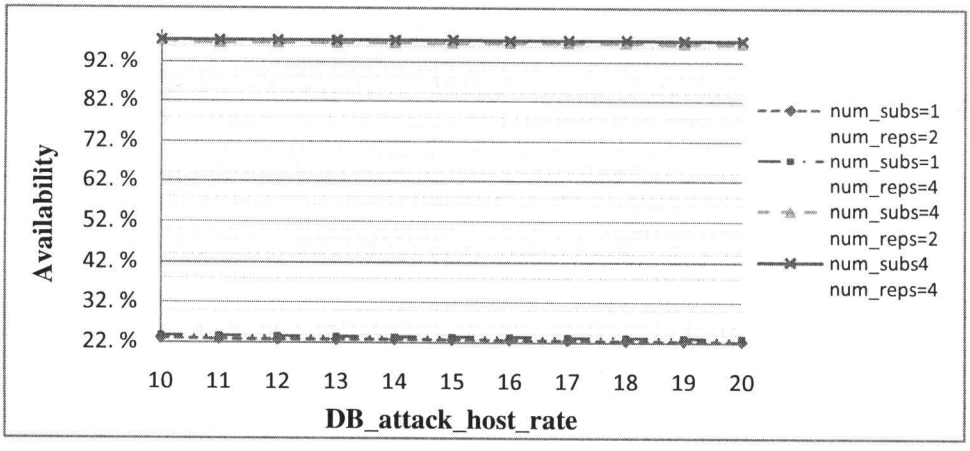
Figure 13- Behaviour of System Security Measures with Respect to Number of Sub-systems and Replicas

As a convention, in this thesis variations in system availability and system integrity are illustrated in graph categories (1) and (2), respectively, which will be presented in the subsequent sections. Each study has a combination of four experiments, which are different in terms of in number of sub-system and replicas in addition to the object variable for a particular study. Different replica and sub-system combinations are listed below:

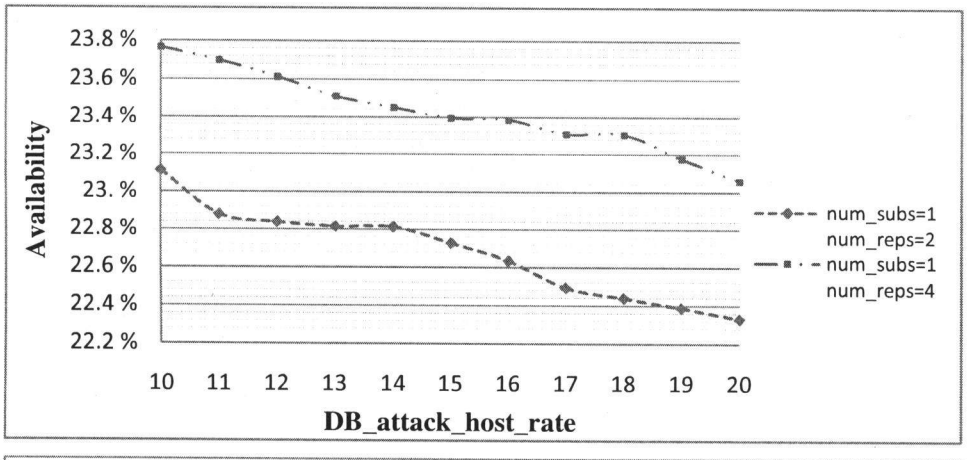
- 1 sub-system with 1 host in each layer and 2 replicas in each host
- 1 sub-system with 1 host in each layer and 4 replicas in each host
- 4 sub-systems with 1 host in each layer and 2 replicas in each host
- 4 sub-systems with 1 host in each layer and 4 replicas in each host

5.2 The Effect of Database Host Attack Rate Changes on Security Measures

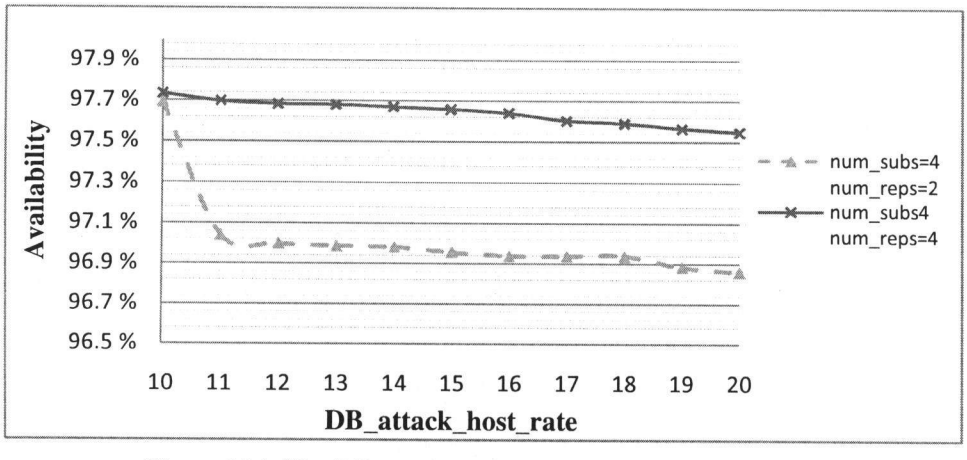
Figure 14.1 and Figure 14.2 illustrate a study where attack rate in data base host increases from 10 to 20 in increments of 1.0. These figures show that both system security measures (i.e. availability and integrity) decrease as the Host Attack rate increases. System security measures tend to improve when increasing the number of replicas and the number of sub-systems. For better illustration, Figure 14.1(b) and Figure 14.1(c) represent closer views of portions of Figure 14.1(a) in which the number of sub-system was chosen to be 1 and 4, respectively. Comparing these two graphs shows that the effect of *DB_Attack_host_rate* is much higher when smaller number of sub-systems constitutes the system, while in both cases an increase in *DB_Attack_host_rate* will lead to a loss in system security measures. It is worth to point out that since an attack event may propagate from the affected host to the corresponding replicas, with no attacks detected, the rate of the selected replica attack is much more than the largest host attack rate. Therefore, increasing the rate of attack against host not only affects host vulnerability, it will also increase the vulnerability of replicas running on the host, and adversely increases the number of failed replicas in each host, and ultimately affecting system availability and integrity by different means. The adverse increase in the number of failed replicas is due to maintaining the same attack detection rate as well as identical rate of replica repair.



14.1(a)

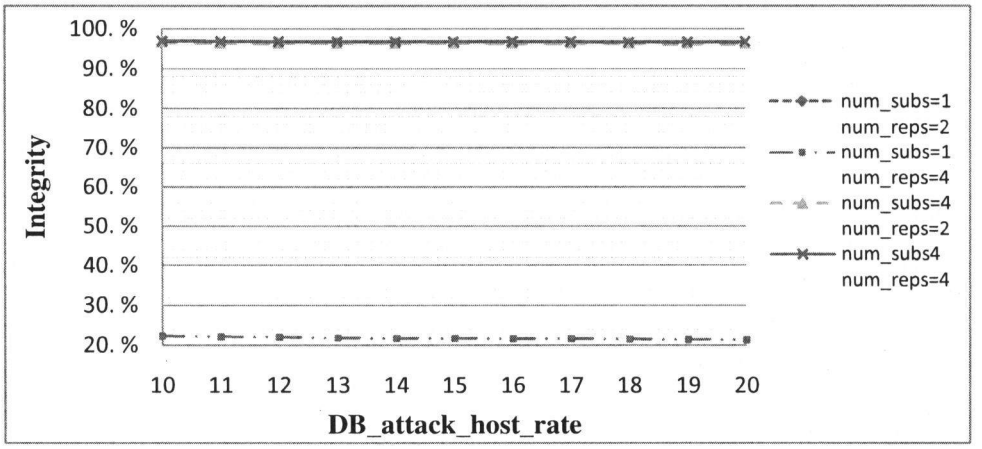


14.1(b)

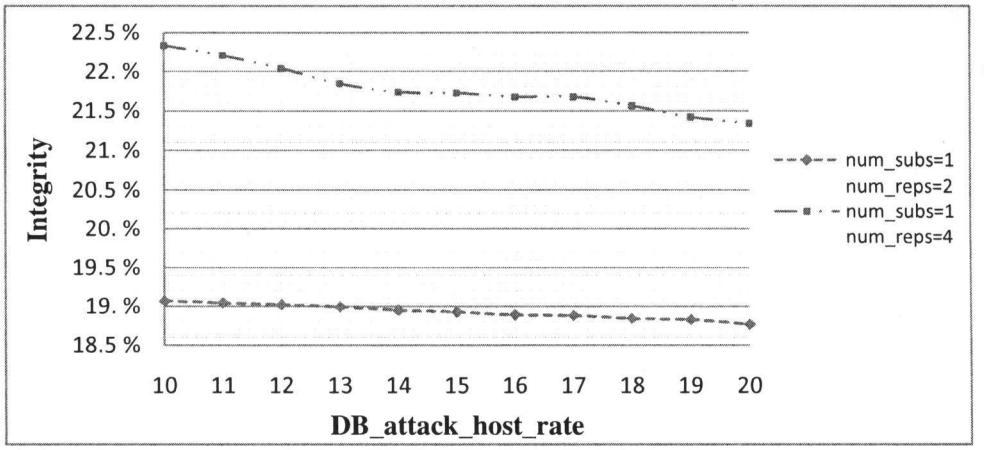


14.1(c)

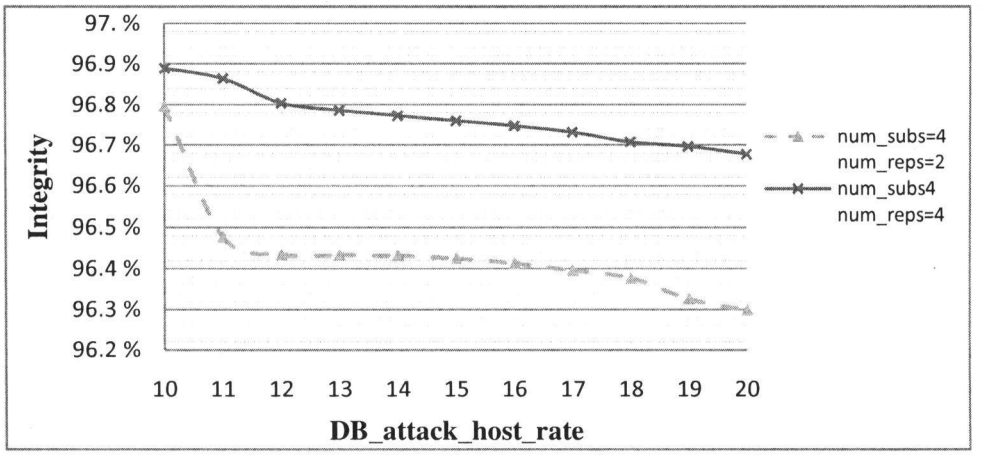
Figure 14.1- The Effects of DB Host Attack Rate changes on Availability



14.2(a)



14.2(b)



14.2(c)

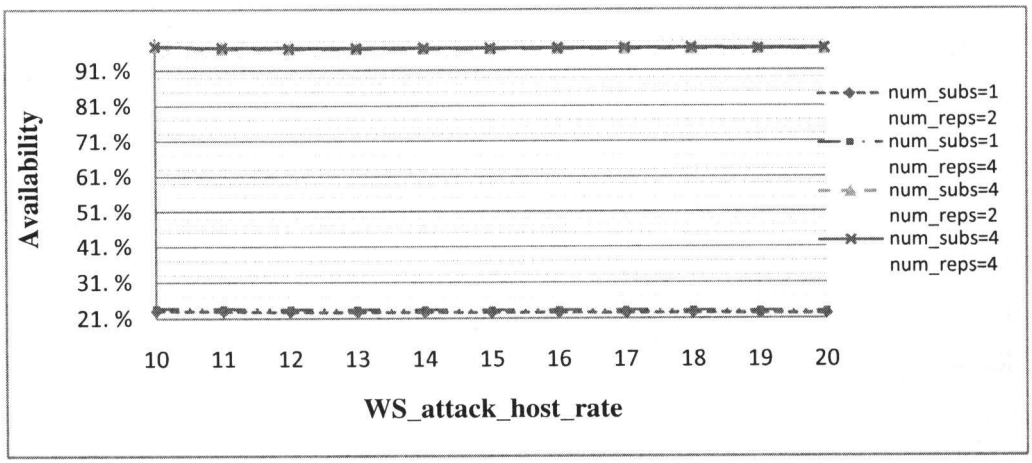
Figure 14.2- The Effects of DB Host Attack Rate changes on Integrity

5.3 The Effect of Web Server Host Attack Rate Changes on Security Measure

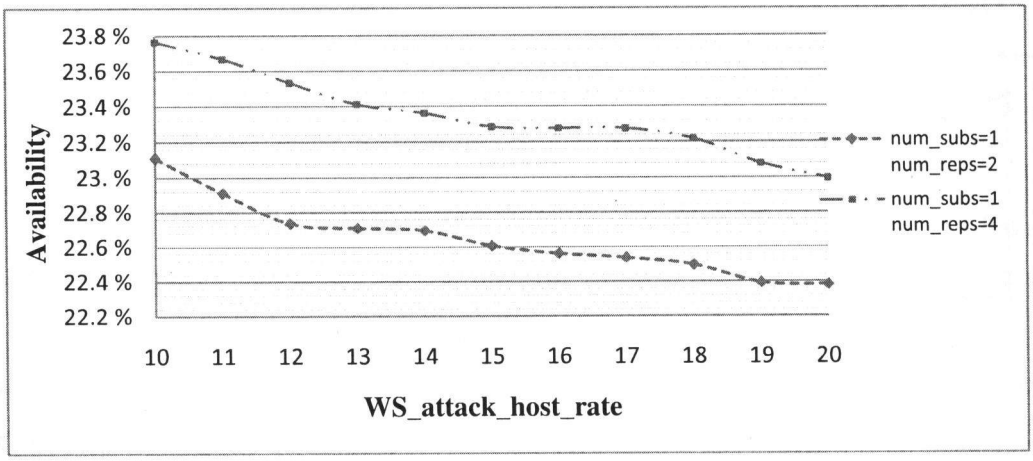
Similar experiments as described in the previous section were performed for the Web Server Host Attack Rate. The results of these experiments also prove similar results as those observed when changing the variable DB_Attack_host_rate. In these experiments, Database host attack rates, replica attack rate in both layers (i.e. host and replica), and the rate of detection of any intrusion in the system are kept constant at values of 10, 40, 5, and 10, respectively. The following graphs, however, were plotted using the results obtained when increasing Web Server Attack Rate from 10 to 20 in increments of 1.0.

It is noted that while the number of sub-system and replica increase, the availability and security measures of the system noticeably convalesce; yet increasing the rate of attack in web server host has the opposite effect in the behaviour of system security measures.

15.1(a)



15.1(b)



15.1(c)

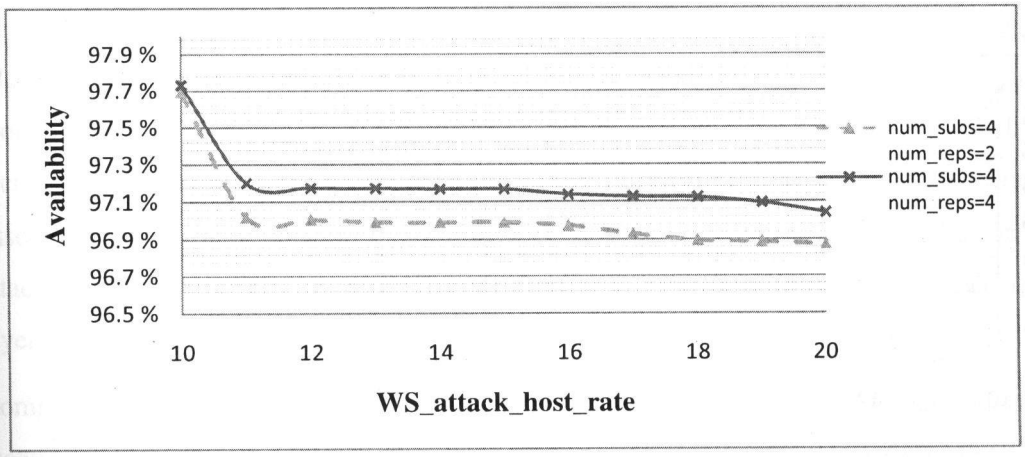
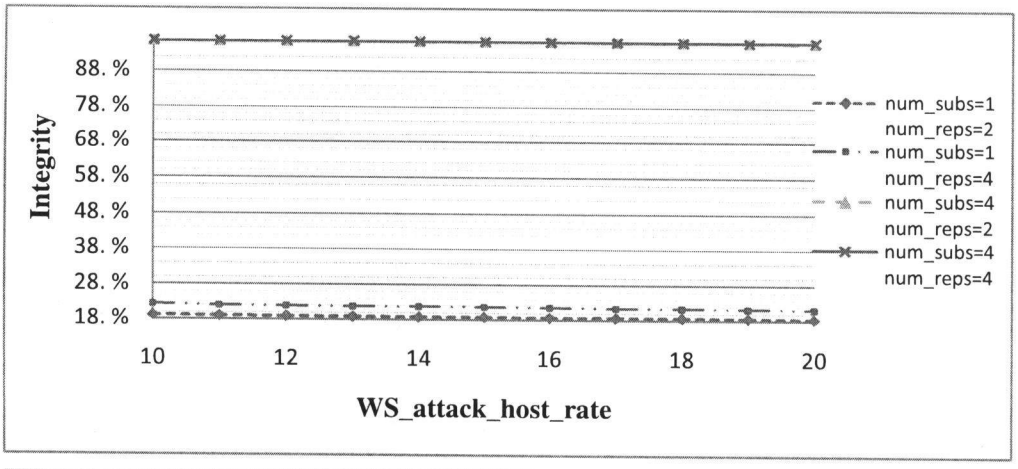
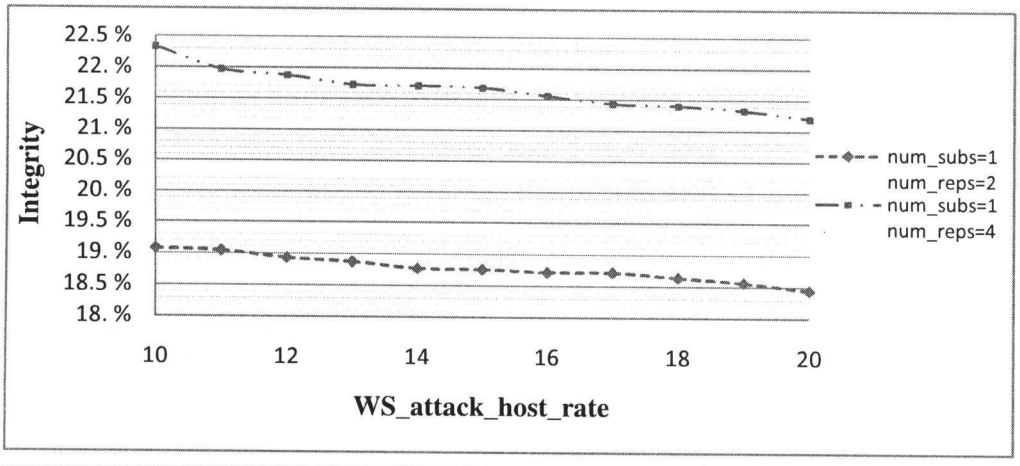


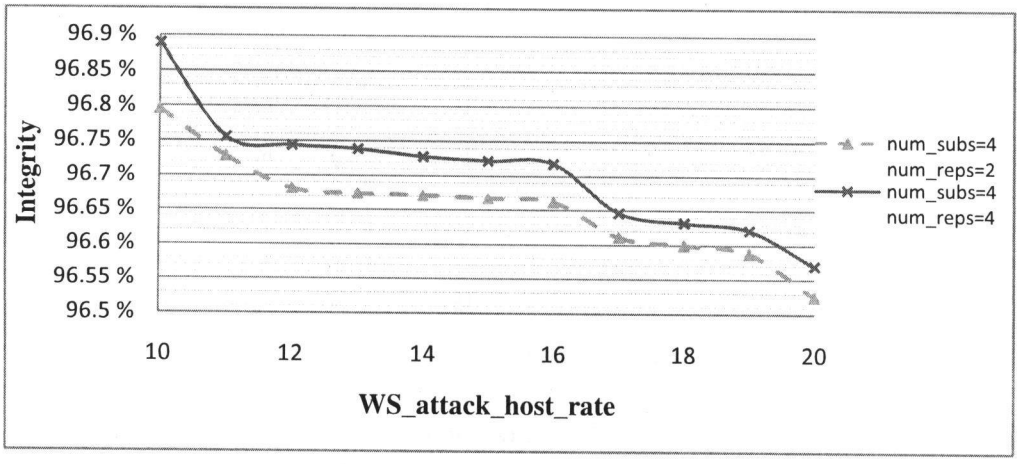
Figure 15.1- The Effects of Web Server Host Attack Rate Changes on Availability



15.2(a)



15.2(b)



15.2(c)

Figure 15.2- The Effects of Web Server Host Attack Rate Changes on Integrity

5.4 Comparison of Rate Changes in Database Host Attack and Web Server Host Attack

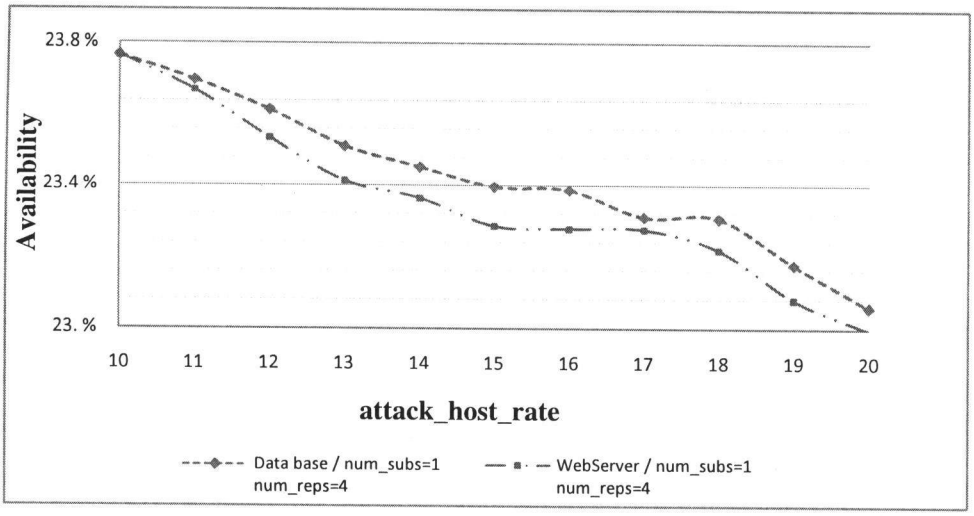
As noted earlier, the main objective of this research has been to evaluate security measures of a layered architecture. Focusing on the results of the conducted experiments, including those obtained from altering the DB_Attack_host_rate and WS_attack_host_rate variables, this section will explore the effects of changes in each layer attack rate on the other layer. Figure 16.1(a) and Figure 16.2(a) show the variation of availability and integrity measures in presence of 1 sub-system and 4 replicas in the system, while the host attack rate in both layers changes from 10.0 to 20.0 in increments of 1.Figure 16.1(b) and Figure 16.2(b) present similar experiments except the number of sub-system that in this case is set to be 4. In each experiment, rate of attack layer is kept constant at 10.0 while the attack rate in the other layer is allowed to change.

When comparing the two plots from part (1) of each experiment, it can be noted that once all the rates, except DB-attack-rate and WS-attack-rate, are kept identical, increasing attack rate in Web Server layer will have a more severe impact on system availability while increasing the attack rate in Database layer will less severely influence system availability. The reason for such behaviours is that while both host attacks (i.e. Web Server and Database) may propagate into all replicas running on the attacked host, presence of any undetected attack in the upper layer (i.e. Web server) also propagates into the lower layer (i.e. Database). The latter propagation causes more vulnerability of the Database layer and consequently decreases system availability. While increase in attack rate in both layers reduces system availability, the upper layer is more susceptible since vulnerability always has a downward propagation and therefore an increase in attack rate for the upper layer will increase the possibility of intrusion existence in the lower layer (i.e. larger reduction in System Availability).

Comparing Section (2) of each graph (i.e. system integrity) will lead to similar results. As earlier mentioned, system integrity is lost when number of undetected hosts targeted with a type of an intrusion becomes more than one third of total number of hosts in the system. In addition, integrity loss may happen when the number of undetected intruded replicas is greater than one third of total number of replicas in the system. When attack rate of Web Server Host increases, all replicas running on that host will have a higher probability of being intruded, while the

quality of intrusion detection software remains unchanged. While undetected attack in Web Server Host propagates into Web Server replicas, there is a possibility of propagation in vulnerability from Web Server replica to its party in the Database layer. Attack propagation is not a possible event upward from the lower layer to the upper layer (although it remains a possibility for an attack event to be broadcasted from database host into its replicas). Therefore, the *Lost_Integrity* place that keeps track of the number of undetected attacked entities will have a higher value when attack rate increases in Web Server layer, which will then result in limited system integrity when upper layer host attack rate is increased for a similar experiment now performed in the lower layer.

16.1(a)



16.1(b)

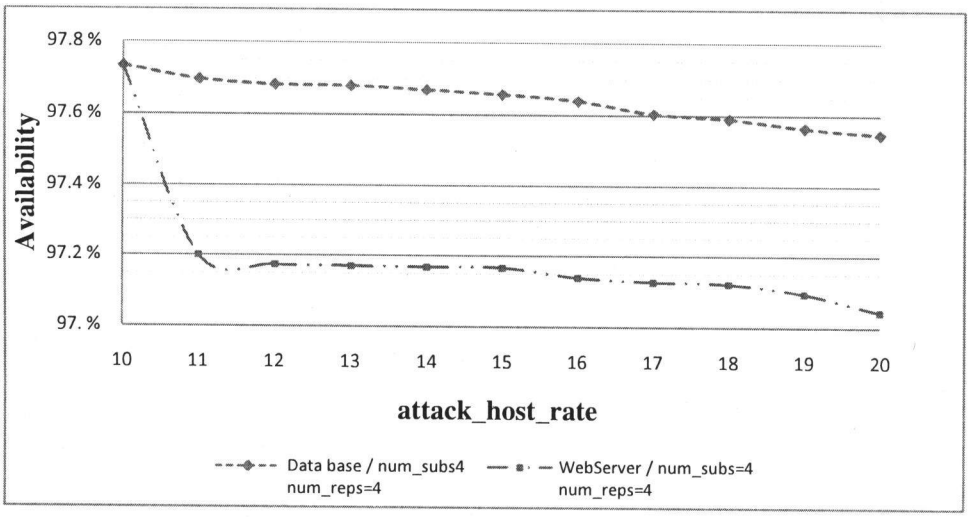
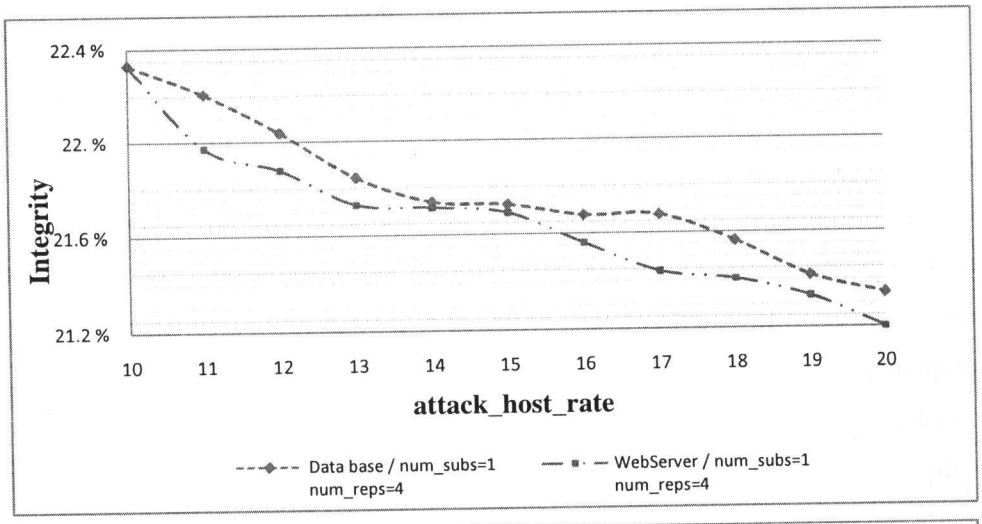


Figure 16.1- Comparison of Rate Changes in DBHost and WSHost Attack (Availability)

16.2(a)



16.2(b)

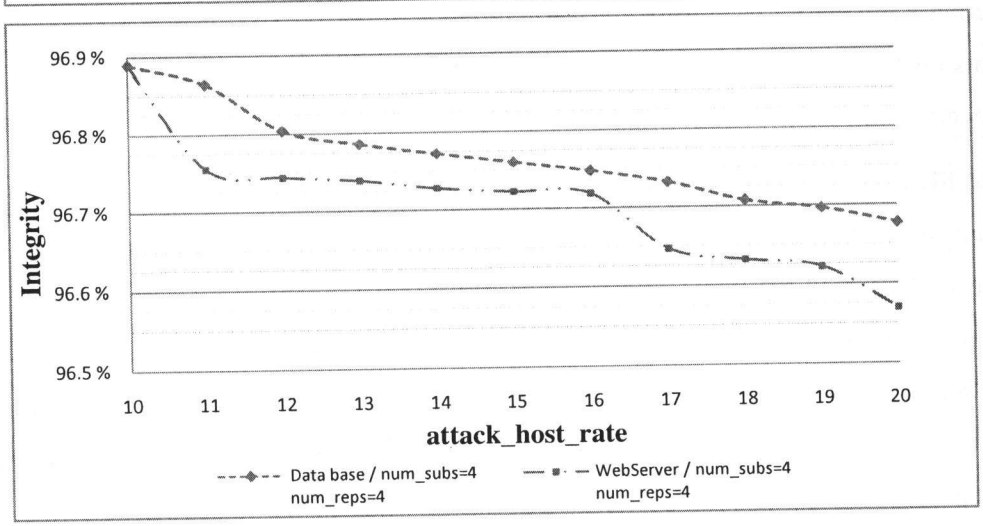


Figure 16.2- Comparison of Rate Changes in DBHost and WS Host Attack (Integrity)

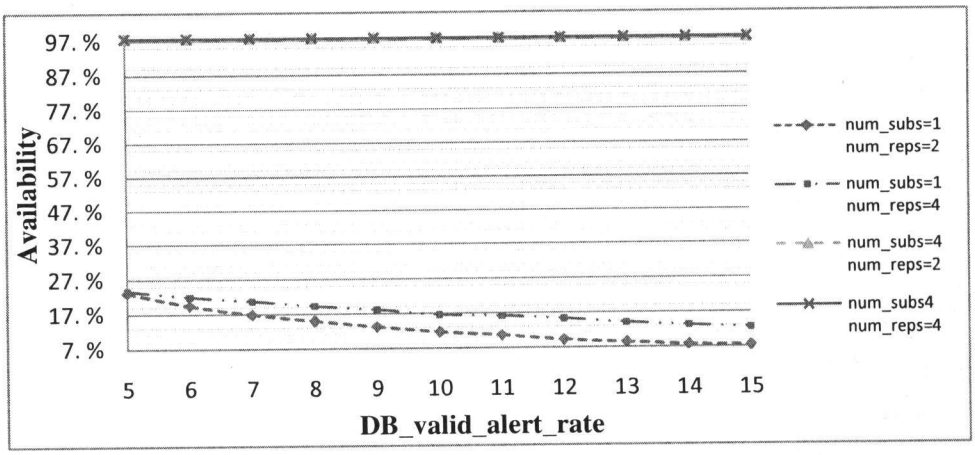
5.5 The Effect of IDSW Quality on Security Measures in Database Host

As noted earlier, almost no intrusion detection software (IDSW) is capable of detecting all the intrusion into a system. High quality and updated IDSW facilitates true detection of an intrusion in a system, yet there always exists a potential for false detection alarm while in reality no system intrusion is in place. To better explore this topic, a set of studies were designed to analyze the impact of the quality of the IDSW on the system security measures. Specifically, we studied the effect of the rate of valid alarms. For these experiments, all the parameters are kept identical to those used in the previous experiments, except that the rate of valid alarms generated by the intrusion detection software on each host was set to vary from 5.0 to 15.0 with 1.0 increment. The value of this range is selected at an extreme for better illustration and such a large value may not be experienced by intrusion detection software.

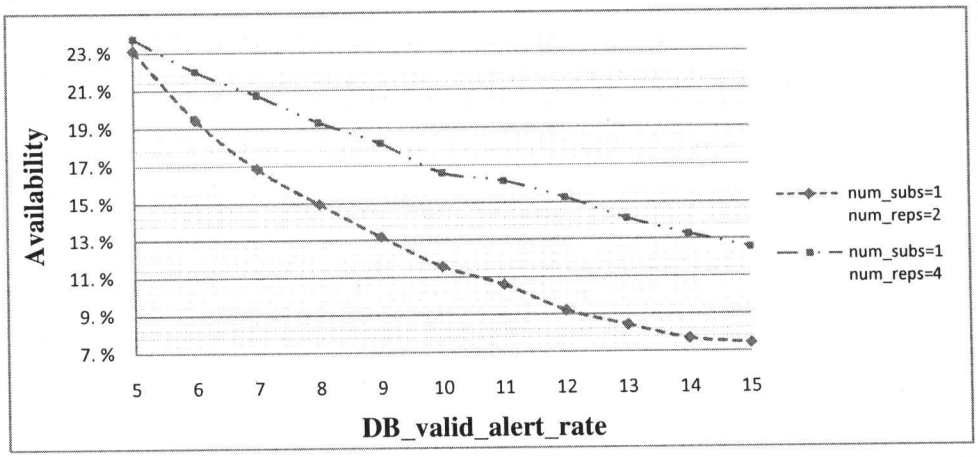
Figure 17.1 and Figure 17.2 respectively illustrate the availability and integrity variations when the quality of intrusion detection software (IDSW) increases. The experiments shown in Figure 17.1 and Figure 17.2 recommend that a higher valid_alert rate (i.e. quality of IDSW) results in a significant improvement in the integrity of the system due to the detection of larger number of attacks in the Database host by the IDSW. Such detection lowers the probability of propagation of an attack from the hosts to the replicas. However, by detecting an attack in the database host, the corrupt host enters into the failed state, resulting in less availability of the system.

Figure 17.1(a) and Figure 17.2(a) illustrate behaviour of a system that includes 1 to 4 sub-systems with 2 and 4 replicas. Figure 17.1(b), 17.1(c), Figure 17.2(b), and Figure 17.2(c) are provided to allow better observation of details of Figure 17.1(a) and Figure 17.2(a) when a design is proposed incorporating 1 and 4 sub-systems along with 2 and 4 replicas in each host.

17.1(a)



17.1(b)



17.1(c)

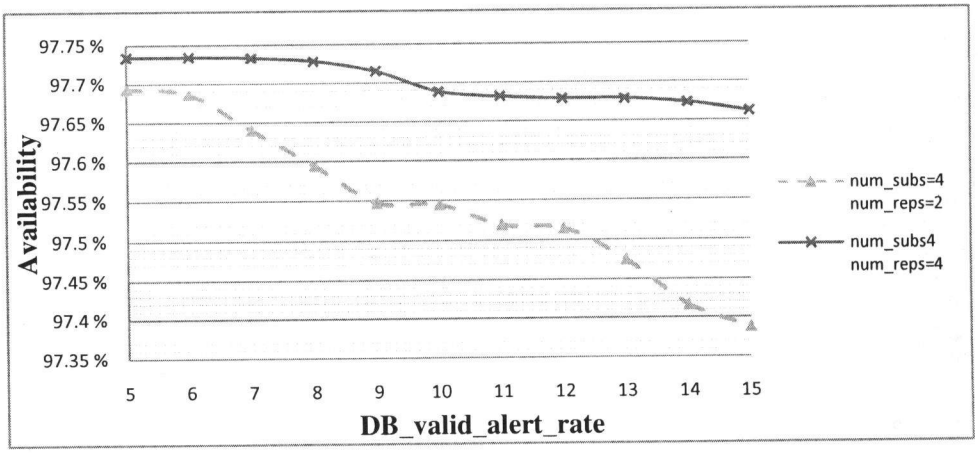
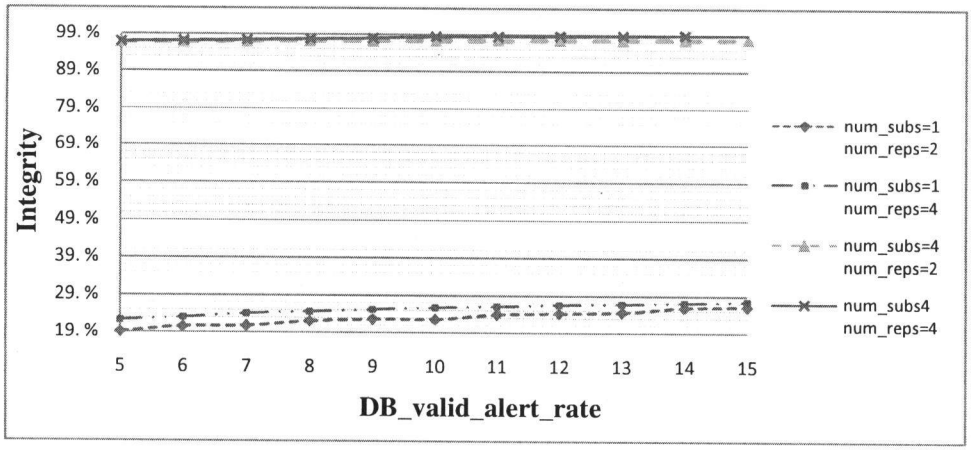
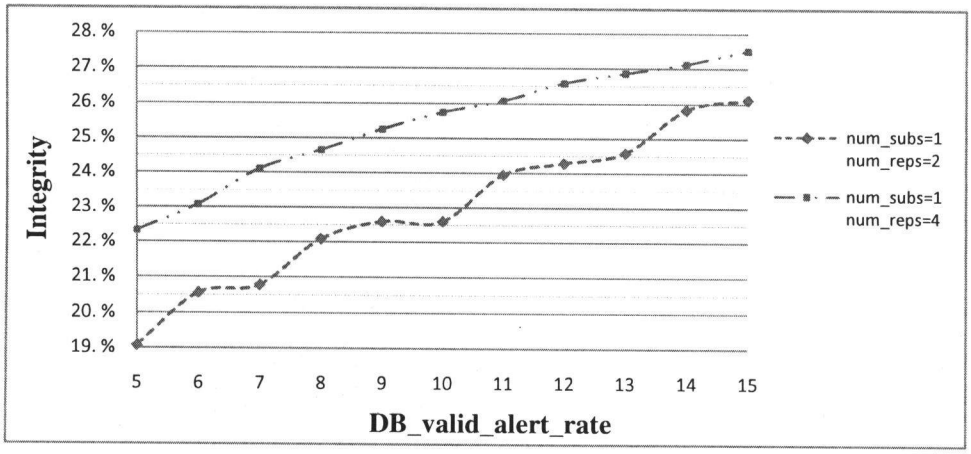


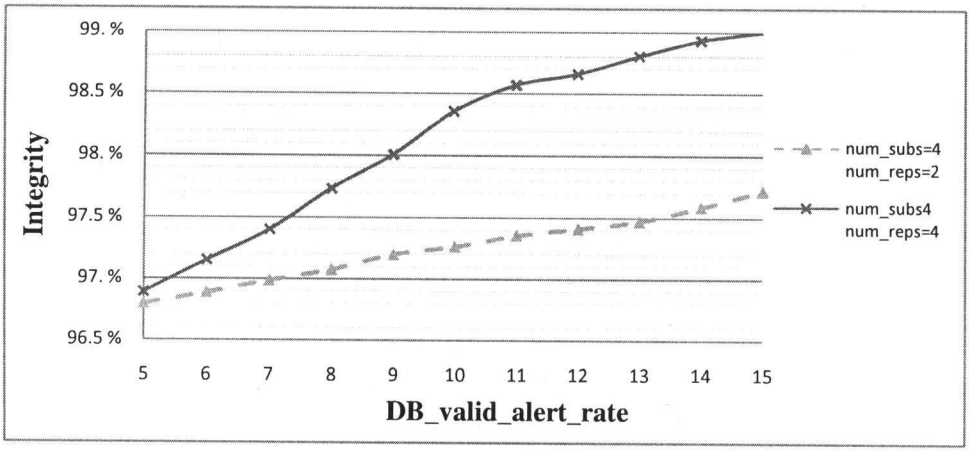
Figure 17.1- The effects of IDSW quality of Database host on Availability



17.2(a)



17.2(b)



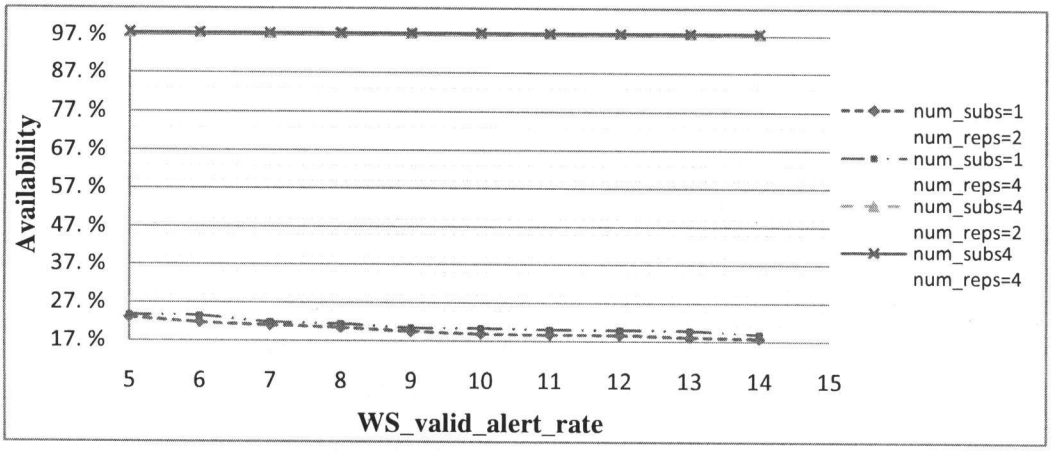
17.2(c)

Figure 17.2- The effects of IDSW quality of Database host on Integrity

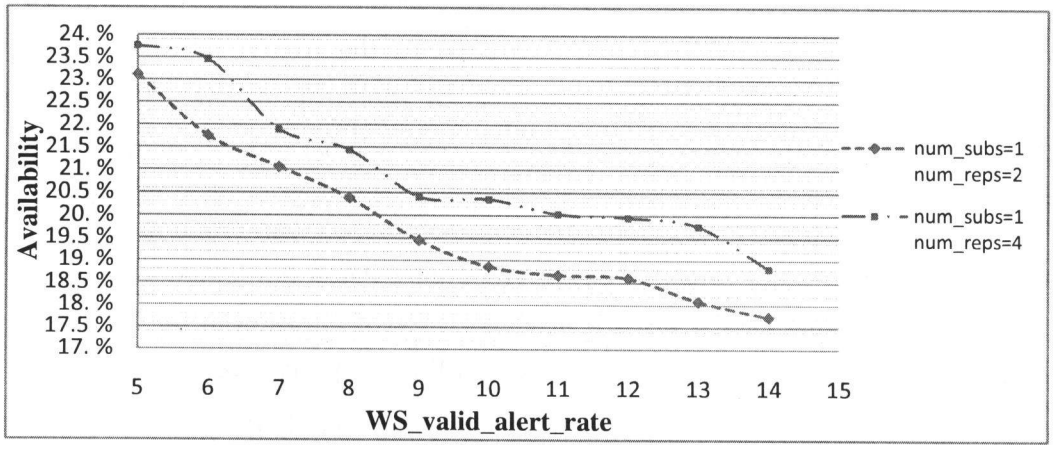
5.6 The Effect of IDSW Quality on Security Measures in Web Server Host

In the foregoing studies, experiments were conducted for system web server host similar to those for system database host. This section describes study the impact of higher quality web server host IDSW when the number of true detection of intrusion increases from 10.0 to 20.0 at 1.0 increment. Figure 18.1(a) and Figure 18.2(a) demonstrate the results for a design experiment containing 1 subsystem, 4 sub-systems, and each host in the sub-system holding 2 and 4 replicas. The portion of 1 and 4 sub-systems are shown in part (b) and (c) of these figures. As can be seen, increasing the value of WS_valid_alert_rate (i.e. increasing number of actual detected attacks) when the attack and repair rates are kept identical, causes a negative effect in the availability of the system. When the web server host's attack is detected by the IDSW, the status of the corrupted host changes to failed mode until it is fully repaired. Moreover, during this period all the replicas running on the corrupt host stop running. Therefore, number of hosts running in the system decreases causing reduction system availability.

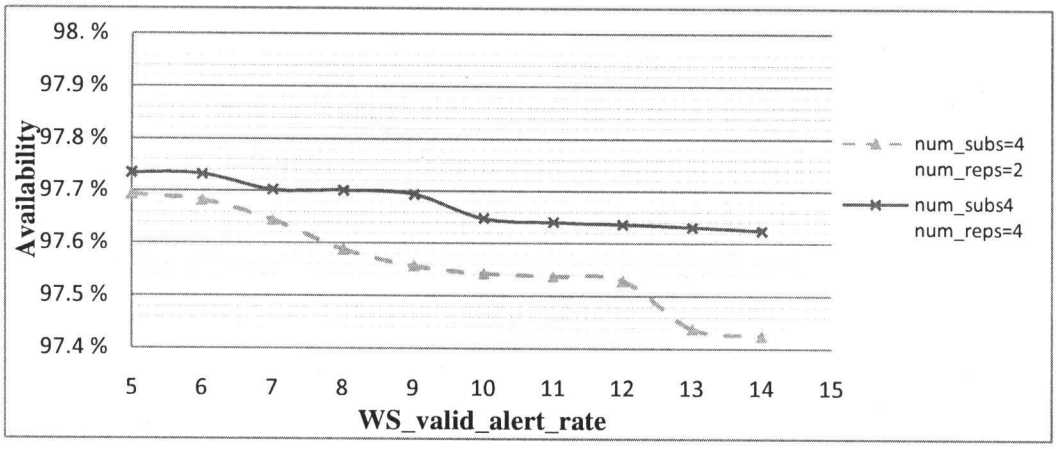
From the integrity point of view, improving the IDSW quality has a positive impact on the system integrity. The system integrity is lost when the presence of intrusion in the host is not detected. During this period (i.e. presence of undetected attack in the host), it is possible that the attack propagates from the web server host into all replicas running on it. Furthermore, the attack may broadcast from an unobserved corrupt replica in the web server layer into its party in the database layer. However, by increasing the value of WS_valid_alert_rate the time interval between occurrence of an attack and its detection becomes shorter. Therefore, not only it causes less time to observe the attack, but also the probability of propagations of attack from web server host into its replicas and also from web server replicas into database replicas are reduced.



18.1(a)

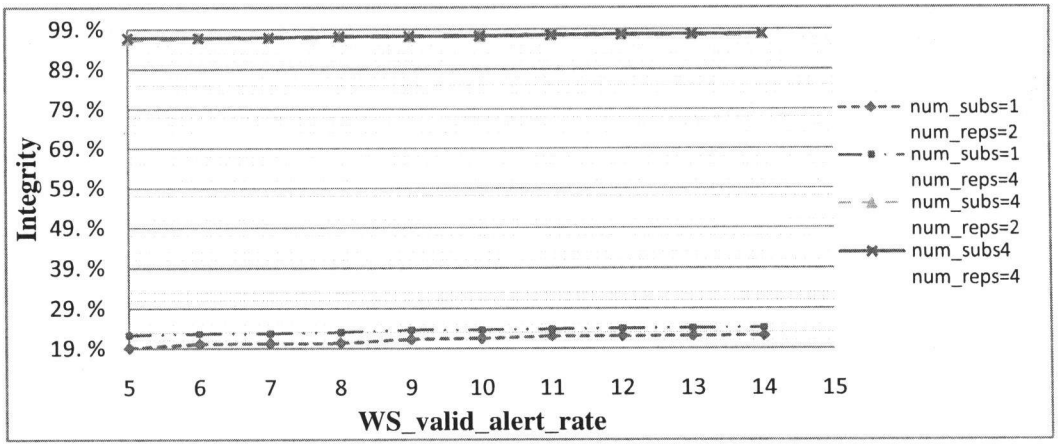


18.1(b)

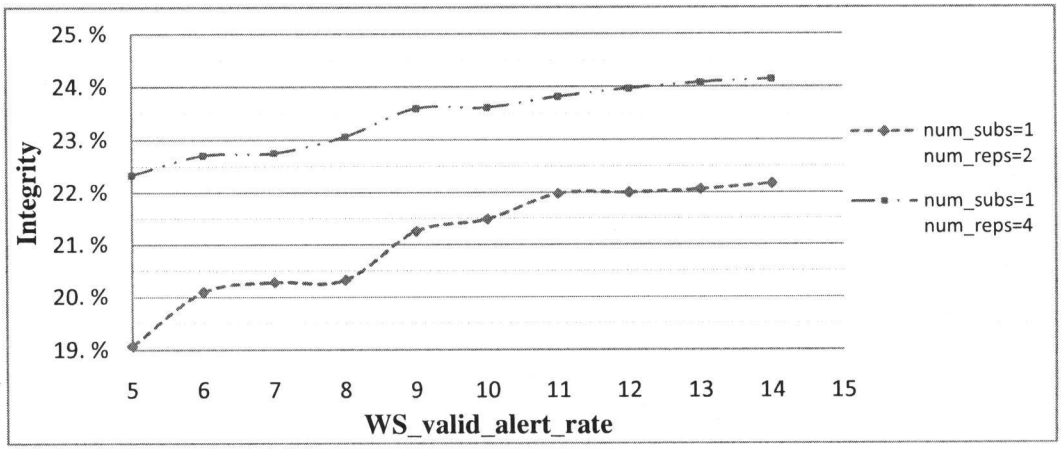


18.1(c)

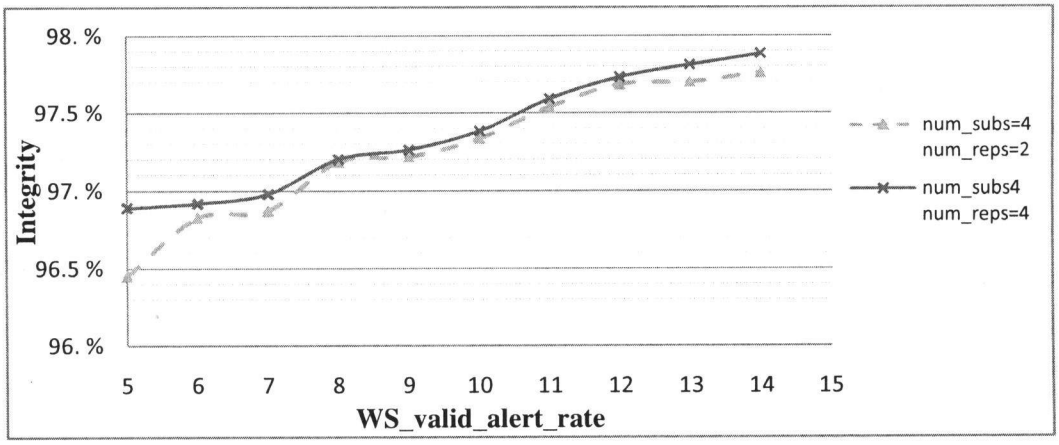
Figure 18.1- The effects of IDSW quality of Web server host on Availability



18.2(a)



18.2(b)



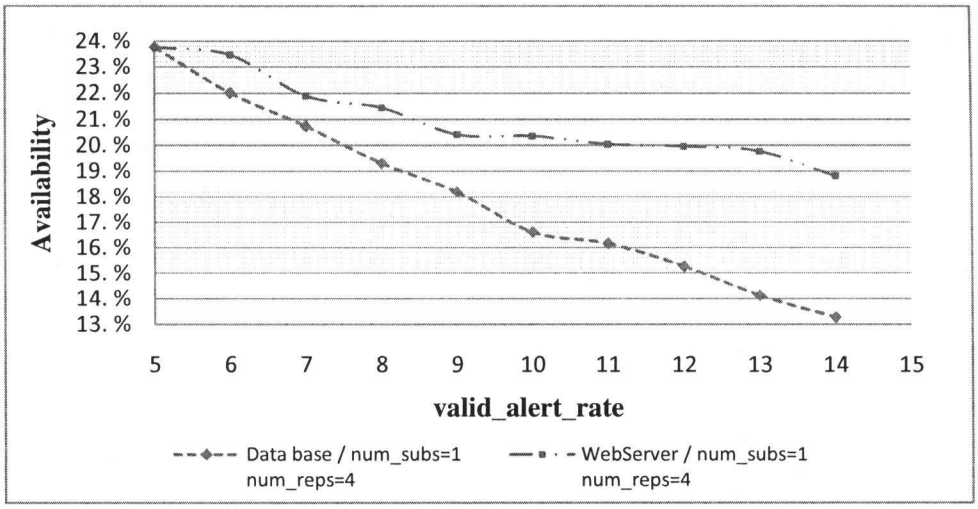
18.2(c)

Figure 18.2- The effects of IDSW quality of Web server host on Integrity

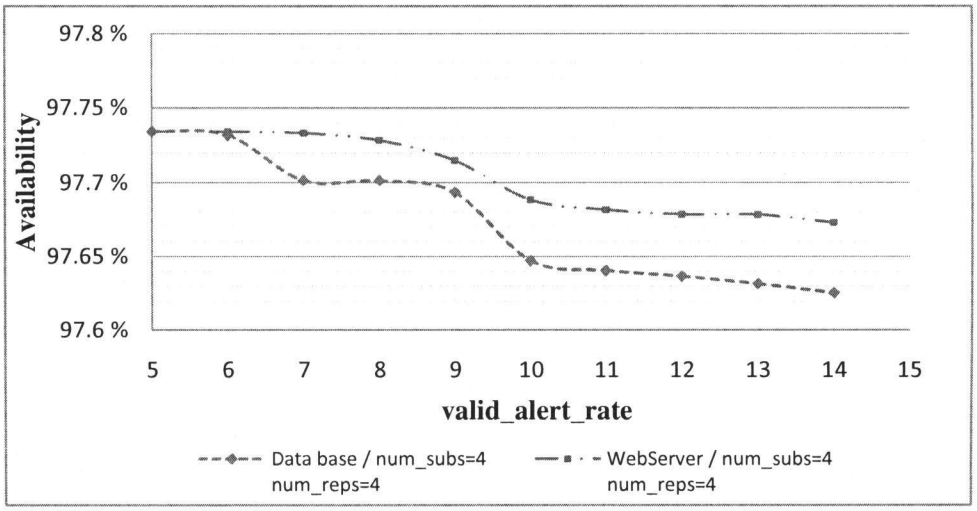
5.7 Comparison of Database Host IDSW Quality and Web Server Host IDSW Quality Changes

In section 5.4 the result of changing host attack rate in one layer was studied against the other layer. This section will study the effect of web server host valid rate enhancements in contrast to those for the database layer. The major results of this study are highlighted in Figure 19.1 and Figure 19.2.

Figure 19.1(a) and Figure 19.1(b) show descending plots for system availability with a design composed of 1 and 4 sub-systems, respectively, allowing for ascending in the value of valid_alert_rate of each host layer. As explained in Sections 5.5 and 5.6, system availability decreases for both layers. Yet, the negative influence of DB_valid_alert_rate (i.e. valid alarm rate of database host) is much more than that of web server layers. Per Section 3.6.2, based on the system architecture status quo, any attack in the system may propagate downward from web server layer through the database layer. Therefore, existence of intrusion in the upper layer increases the vulnerability of the lower layer in the system. The effect of intrusion occurrence is system unavailability that spreads upward from database layer to web server layer; thus, detection of attack in the lower layer causes the database host to enter a failed mode, since database host will not be able to respond to the web server request, resulting in less availability of the system.

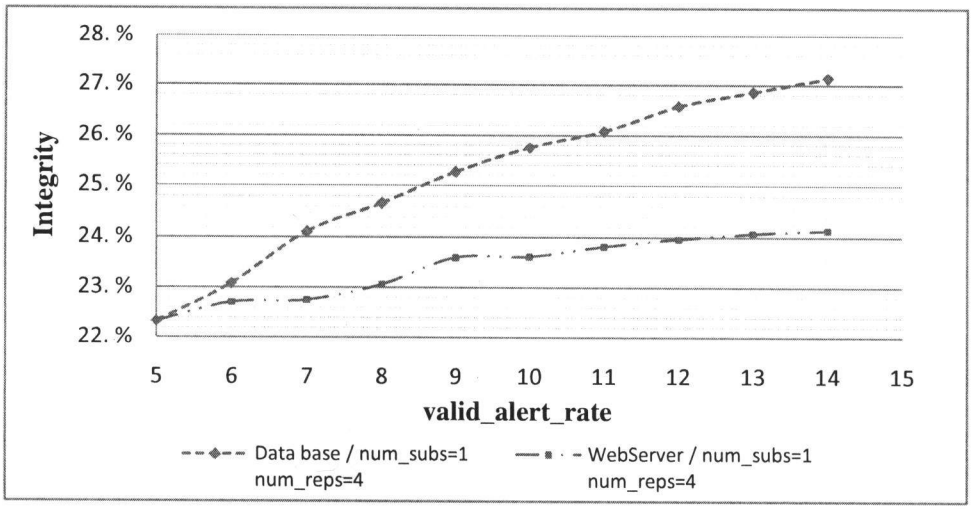


19.1(a)

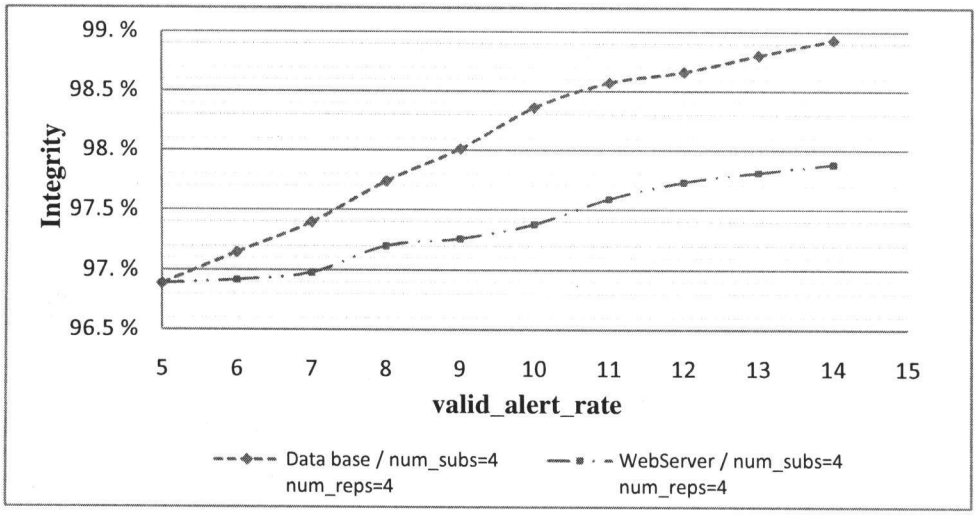


19.1(b)

Figure 19.1- Comparison of DB Host and WS Host IDSW quality Changes (Availability)



19.2(a)



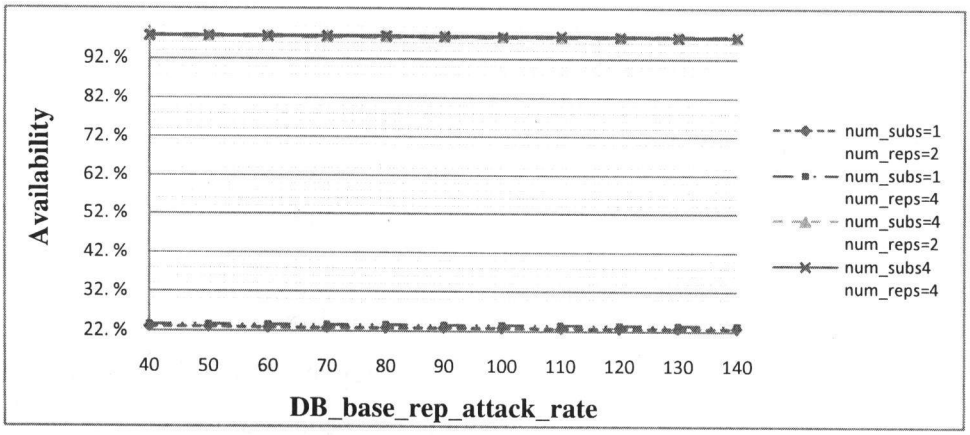
19.2(b)

Figure 19.2- Comparison of DB Host and WS Host IDSW quality Changes (Integrity)

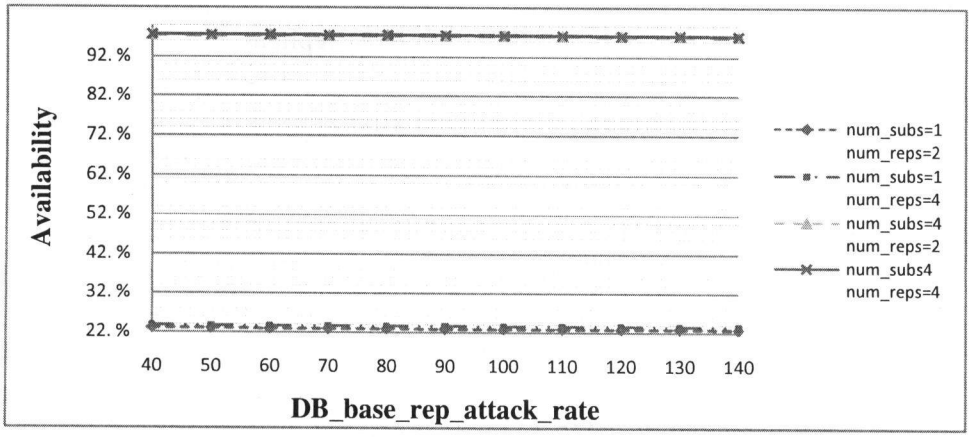
The foregoing studies, designed for specific set of sub-systems, hosts, and replicas, illustrated gradual changes in security measures when both attack rate and valid detection rate for a **host** in each layer were allowed to increase.

A second set of studies have been conducted to determine the behaviour of the system by monitoring the changes in the security measures when attack rate and valid detection rate of a **replica** in each layer have been increased. The remainder of graphs in this chapter will illustrate such experiments. Graph categories (1) and (2) have been reserved to represent the variation in availability and integrity of the system when replica's attack rate and replica's valid alert rates were increased. Note that high-ending values of this range of values is rather extreme and may not be experienced by a real system. These values have rather been exaggerated for better illustration of availability and integrity variations caused by replica attack. As can be seen, similar outcomes have been generated when monitoring the behaviour of system security measures by allowing changes in attack rate and valid alarm rate of replicas.

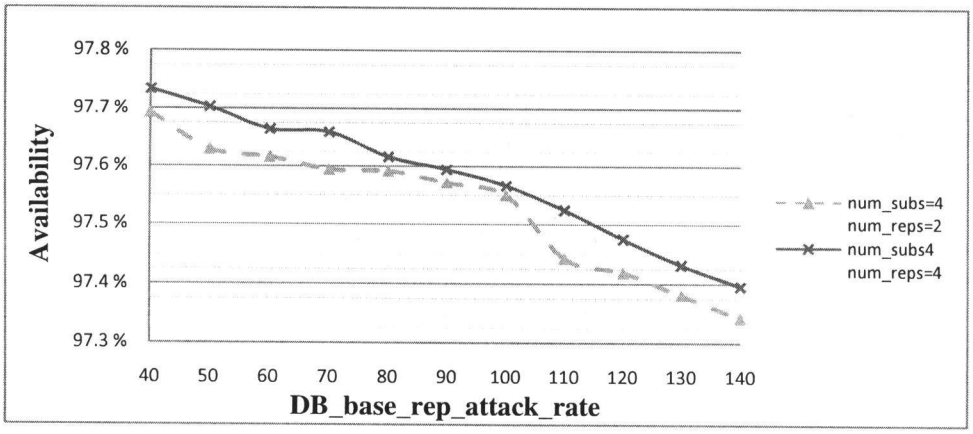
5.8 The Effect of Database Replica Attack Rate Changes on Security Measure



20.1(a)

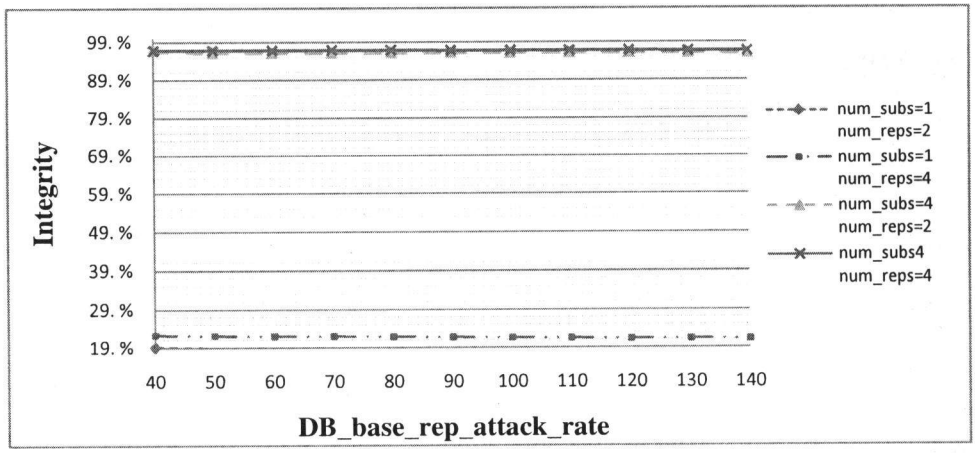


20.1(b)

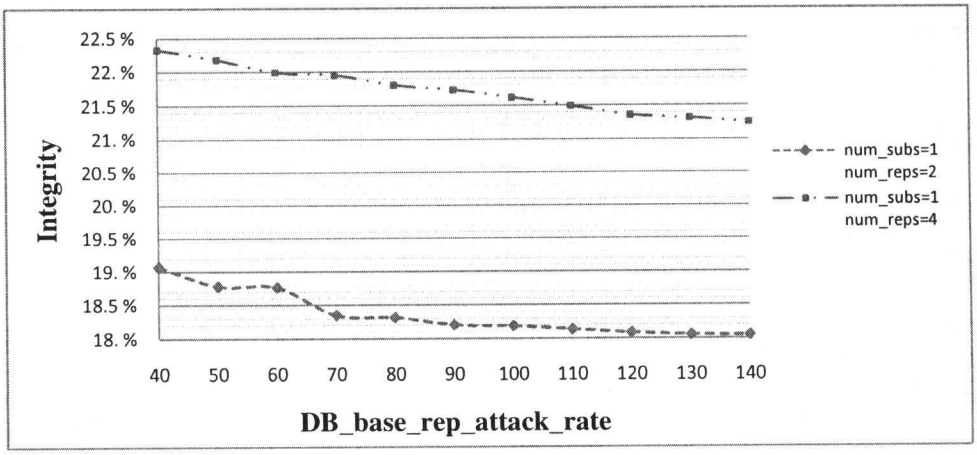


20.1(c)

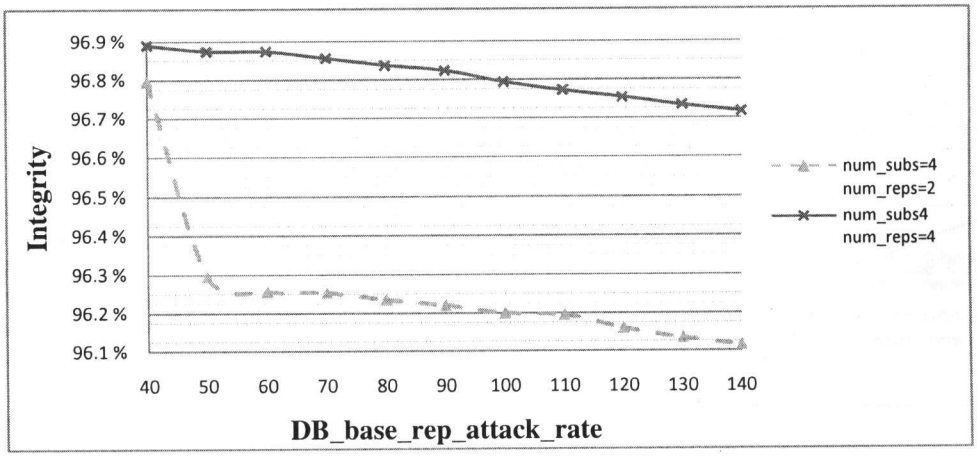
Figure 20.1- Database Replica Attack Rate Effects on Availability



20.2(a)



20.2(b)



20.2(c)

Figure 20.2- Database Replica Attack Rate Effects on Integrity

5.9 The Effect of Web server Replica Attack Rate Changes on Security Measure

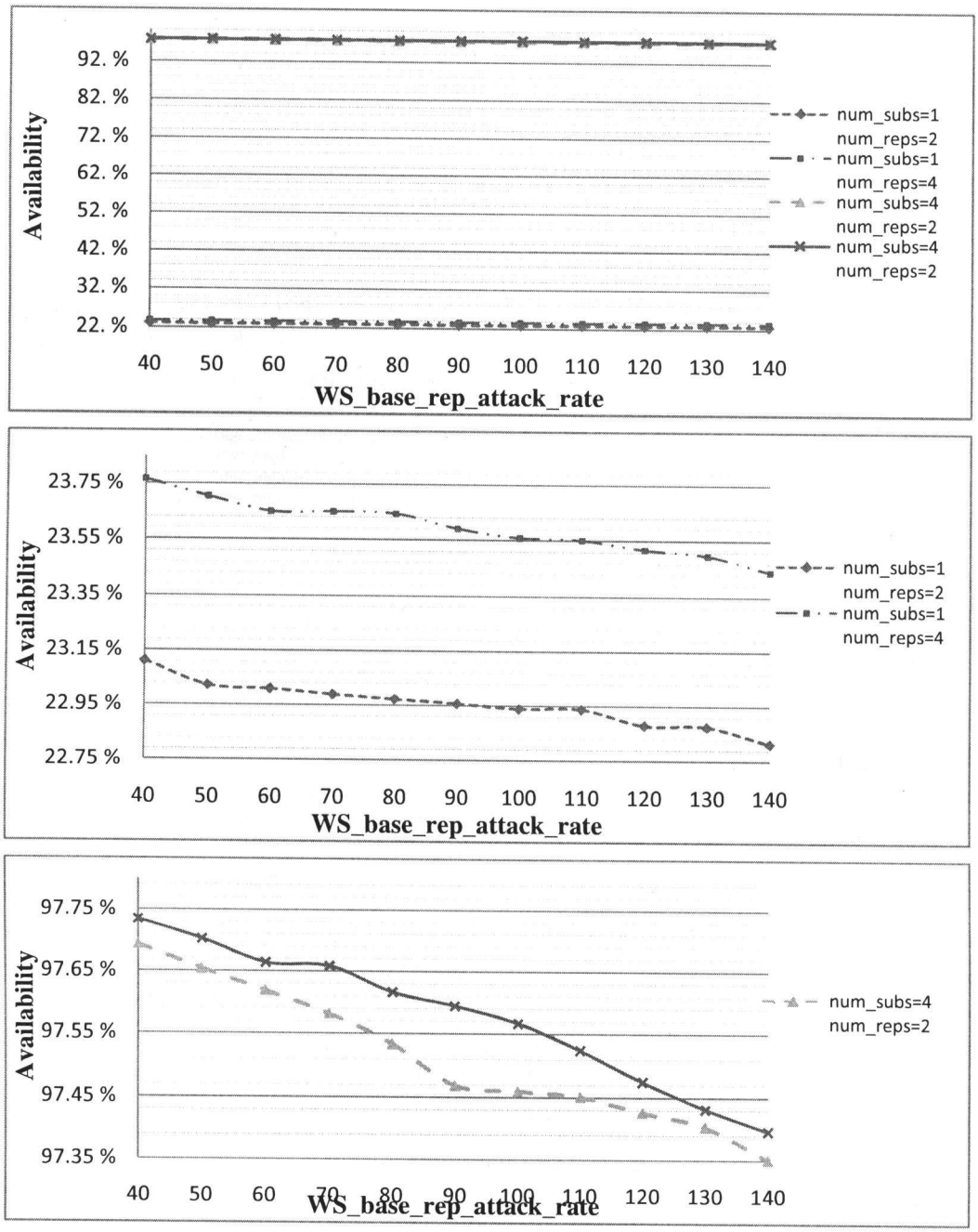
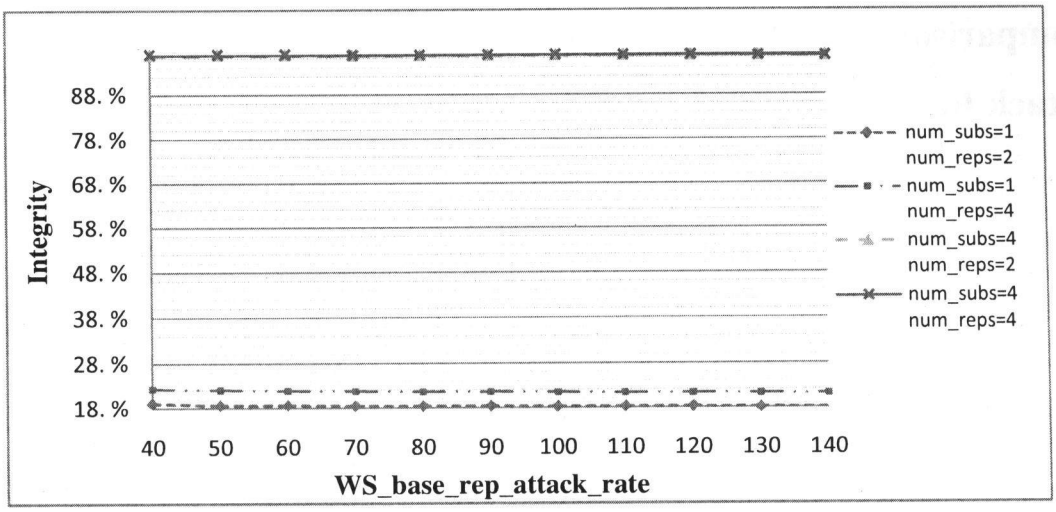
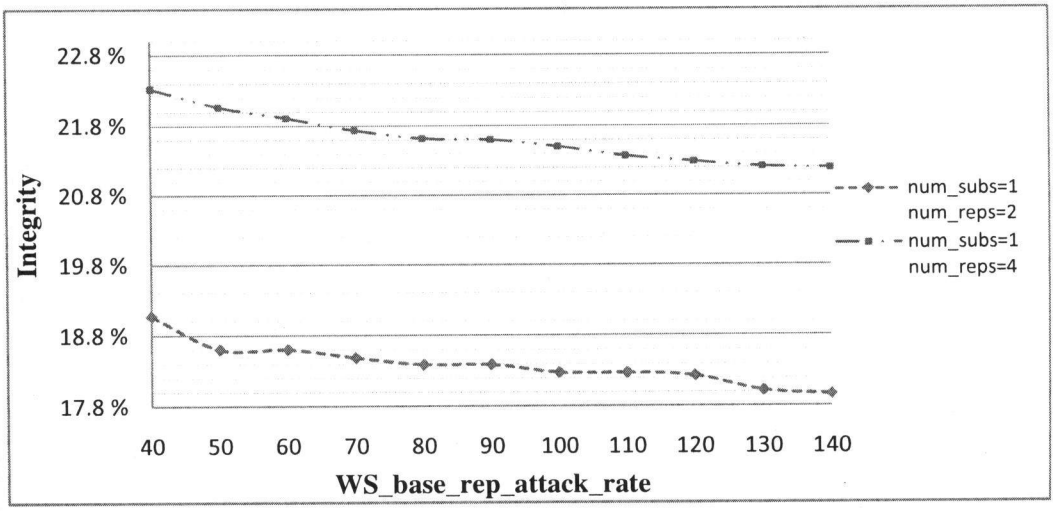


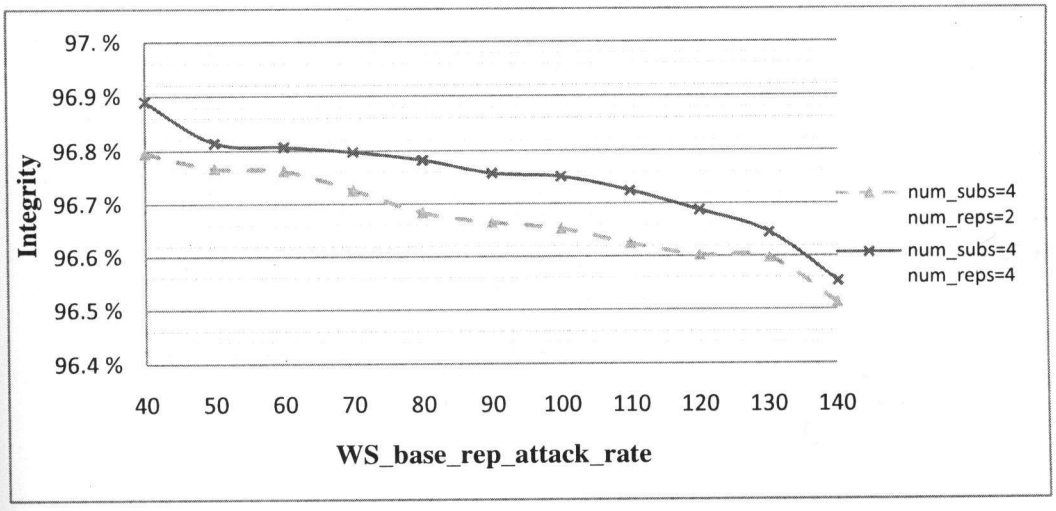
Figure 21.1- Web Server Replica Attack Rate Effects on Availability



21.2(a)



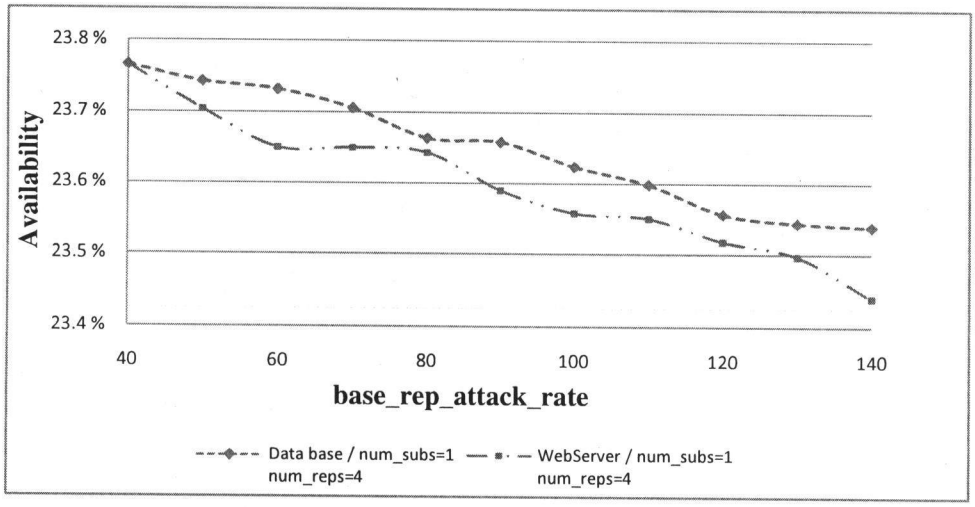
21.2(b)



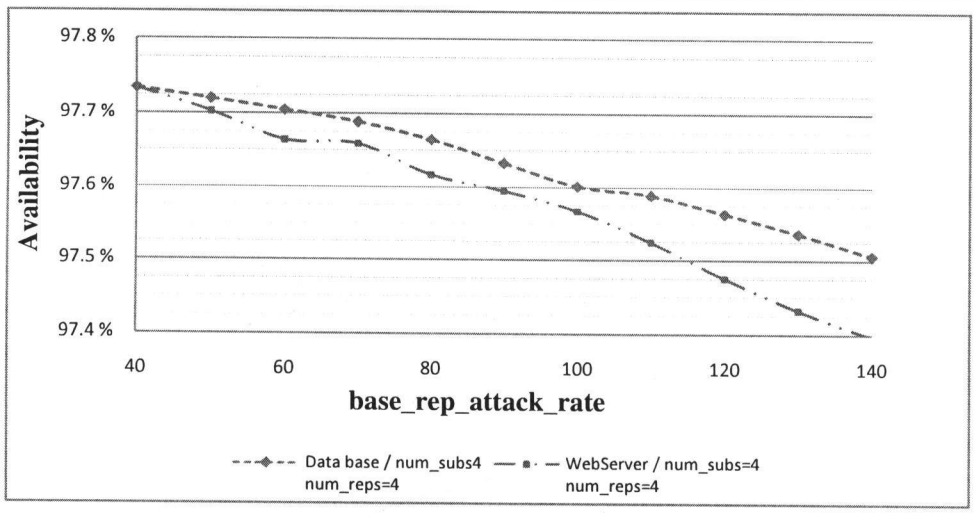
21.2(c)

Figure 21.2- Web Server Replica Attack Rate Effects on Integrity

5.10 Comparison of Database Replica Attack and Web Server Attack Rate Changes



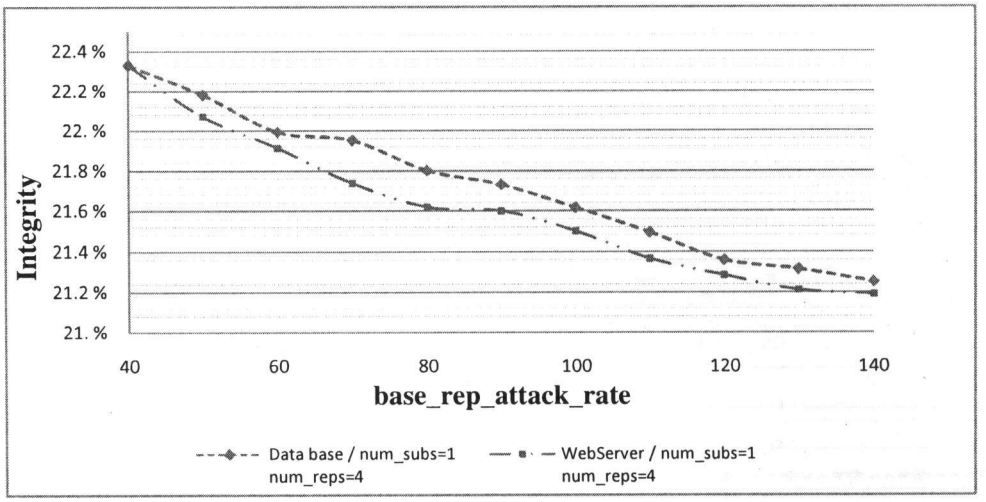
22.1(a)



22.1(b)

Figure 22.1- Comparison of DB Replica and WS Replica Attack Rate Changes (Availability)

22.2(a)



22.2(b)

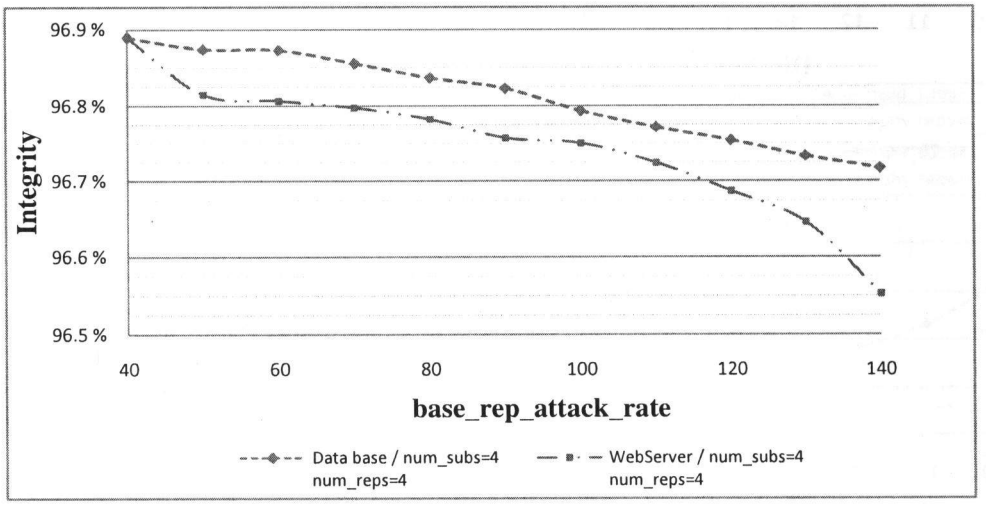
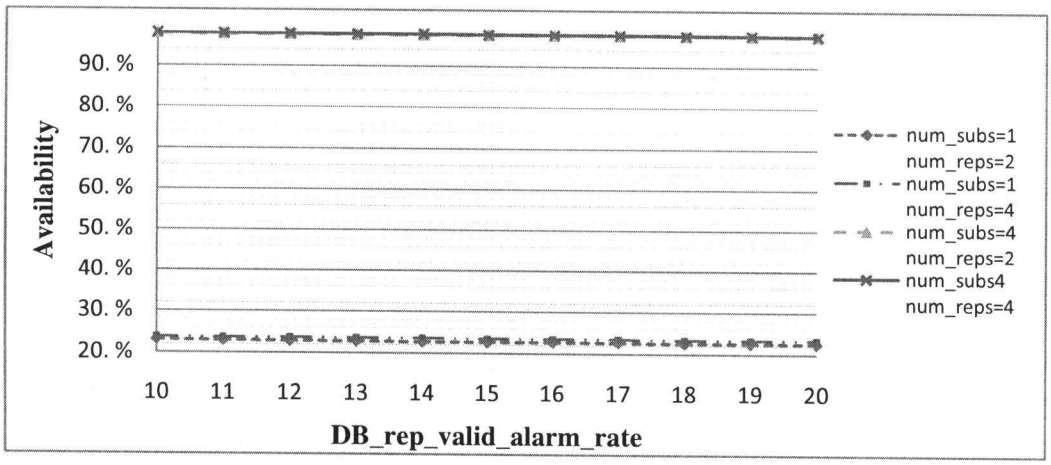
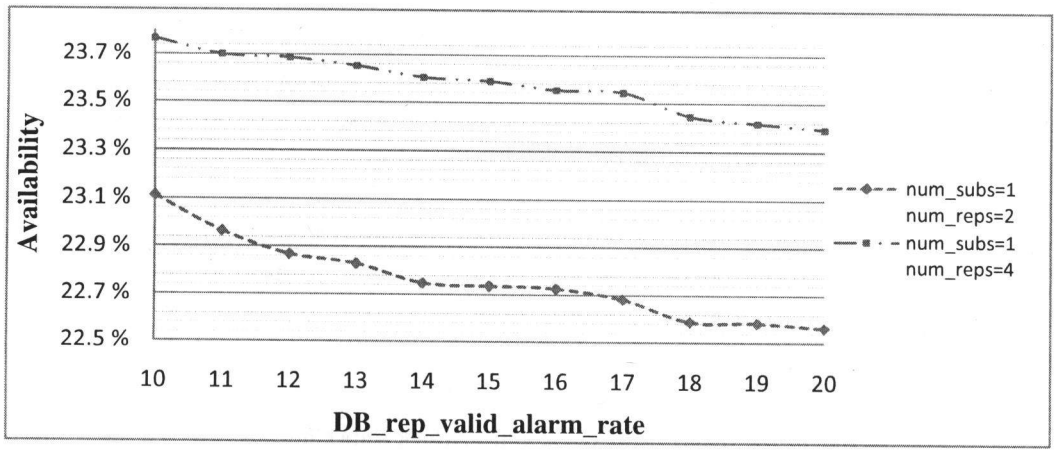


Figure 22.2- Comparison of DB Replica and WS Replica Attack Rate Changes (Integrity)

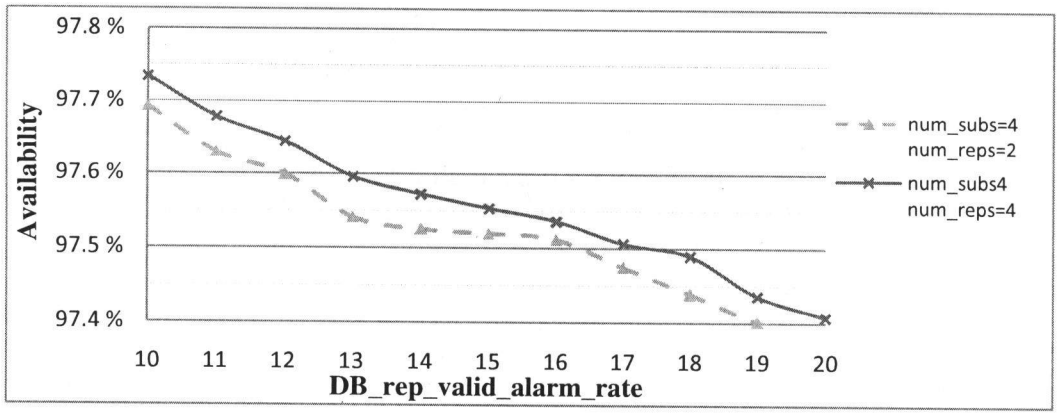
5.11 The Effect of IDSW Quality on Security Measures in Data base Replica



23.1(a)

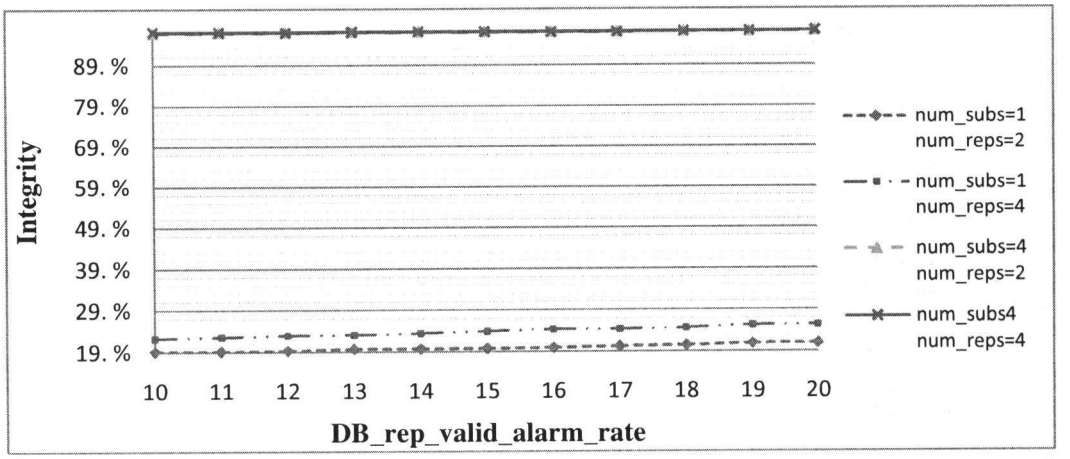


23.1(b)

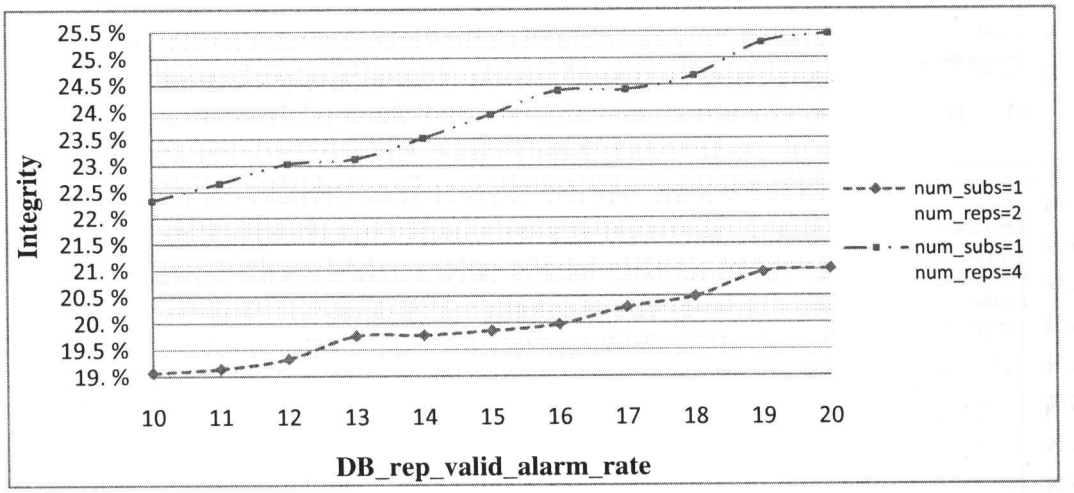


23.1(c)

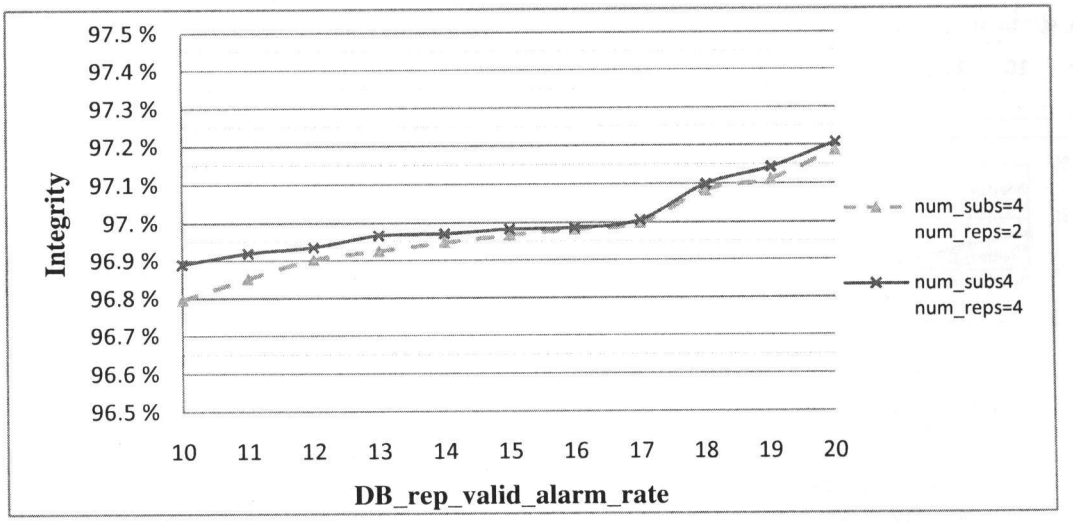
Figure 23.1- The effects of IDSW quality of Database replica on Availability



23.2(a)



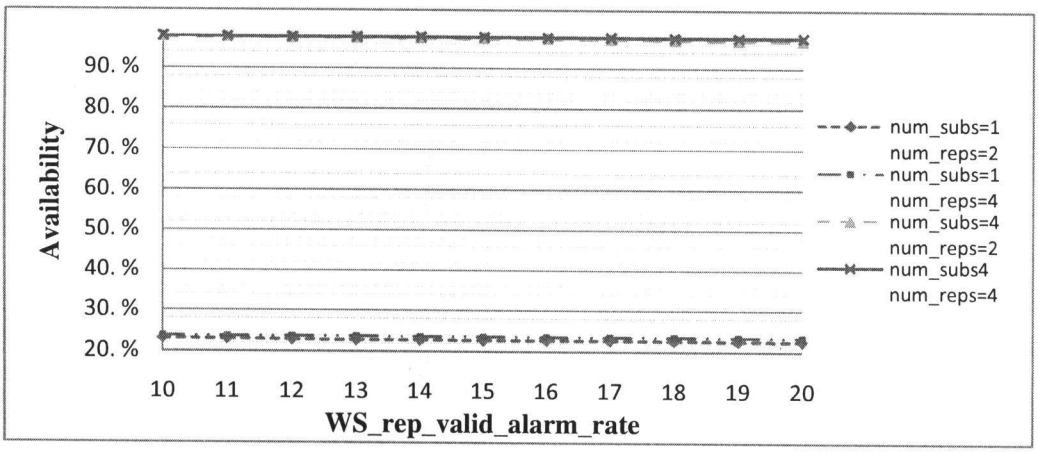
23.2(b)



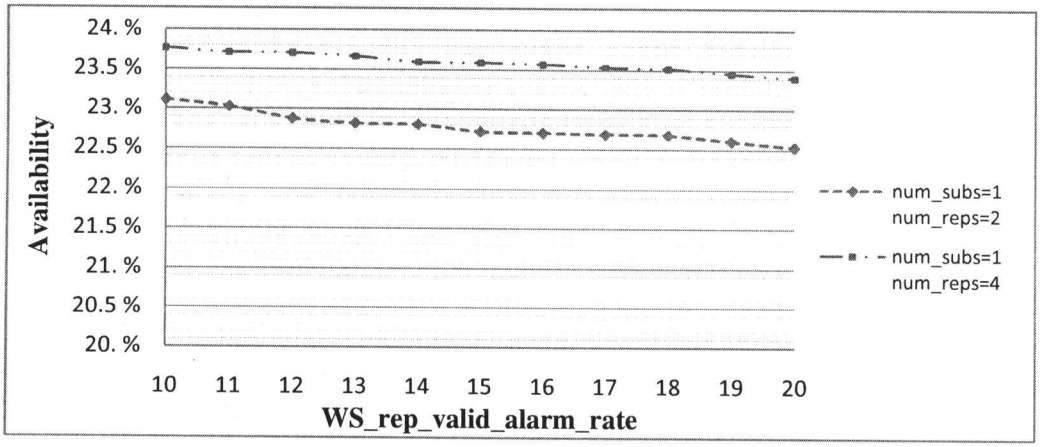
23.2(c)

Figure 23.2- The effects of IDSW quality of Database replica on Integrity

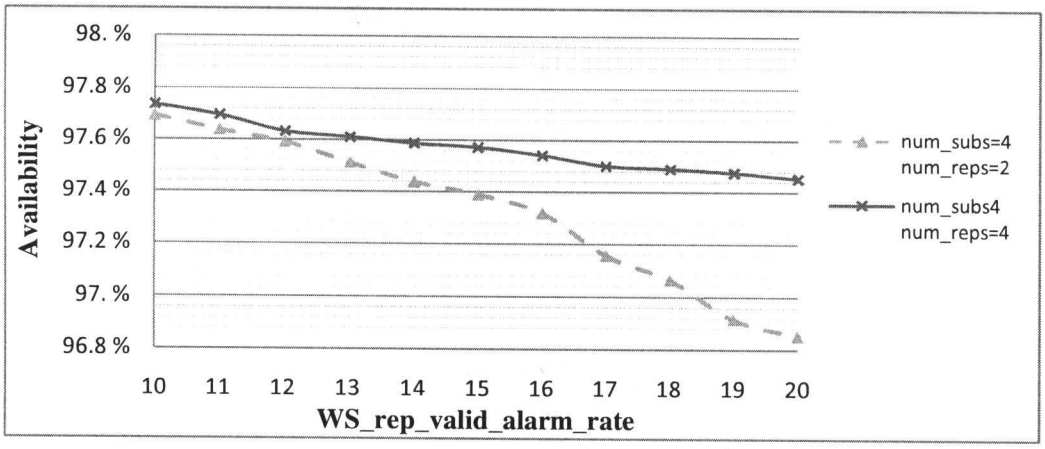
5.12 The Effect of Changing Quality of IDSW in Web server Replica on Security Measures



24.1(a)

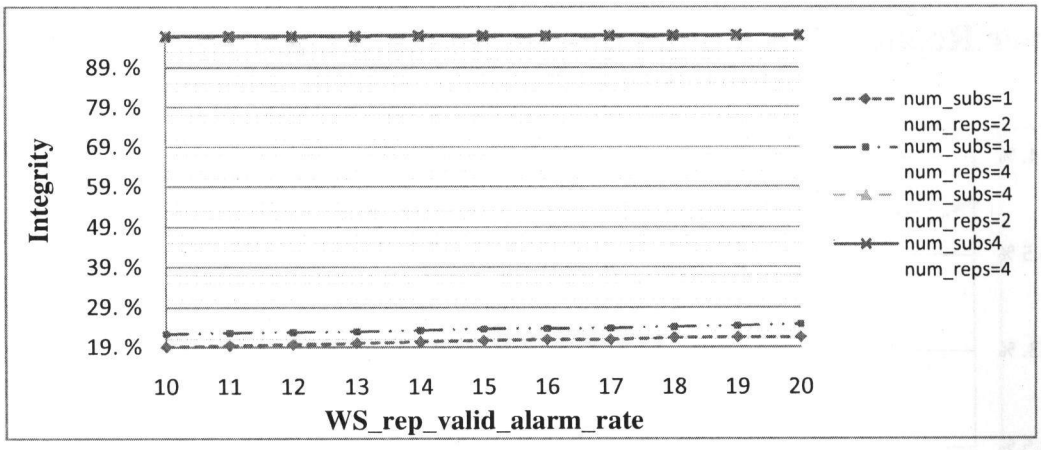


24.1(b)

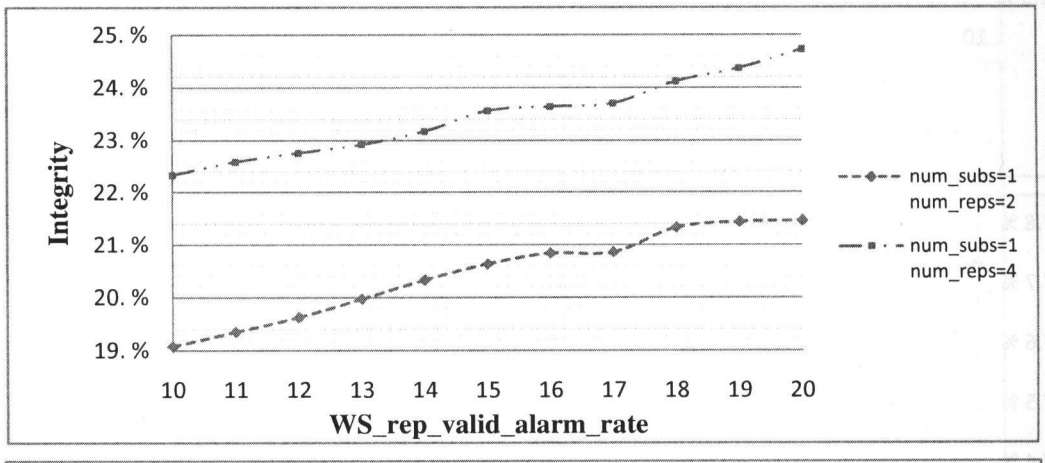


24.1(c)

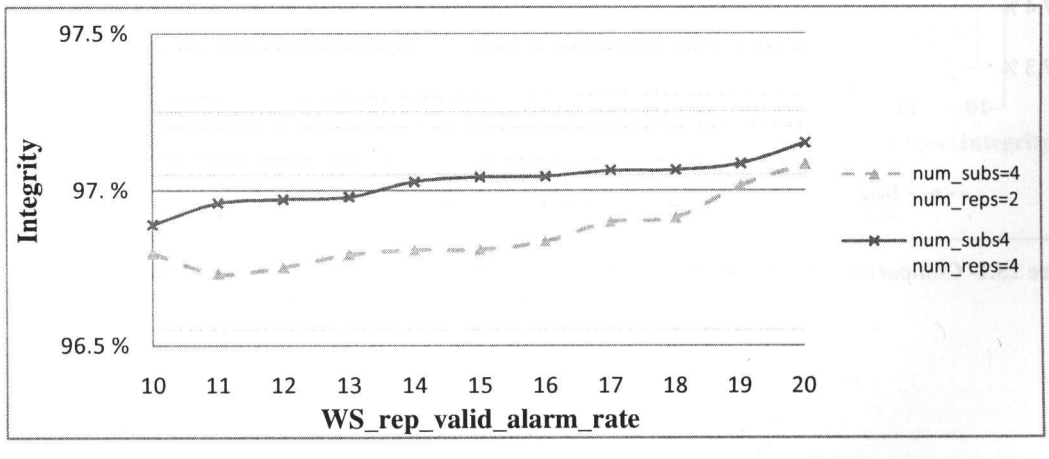
Figure 24.1- The effects of IDSW quality of Web server replica on Availability



24.2(a)



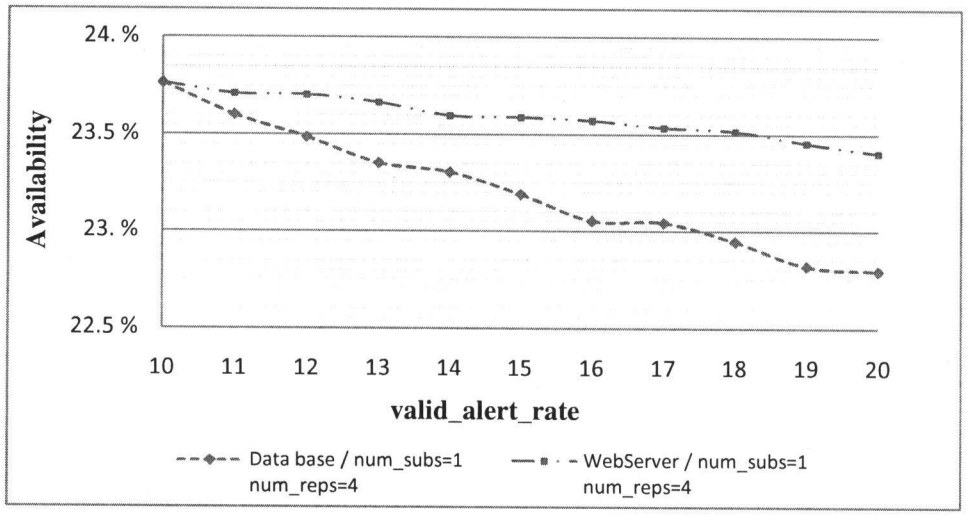
24.2(b)



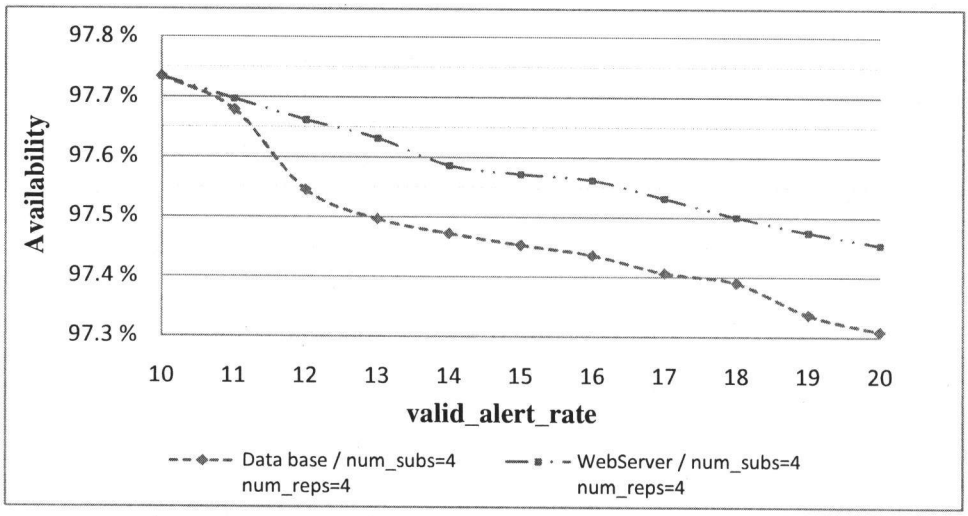
24.2(c)

Figure 24.2- The effects of IDSW quality of Web server replica on Integrity

5.13 Comparison of Database Replica IDSW Quality and Web Server Replica IDSW Quality Changes

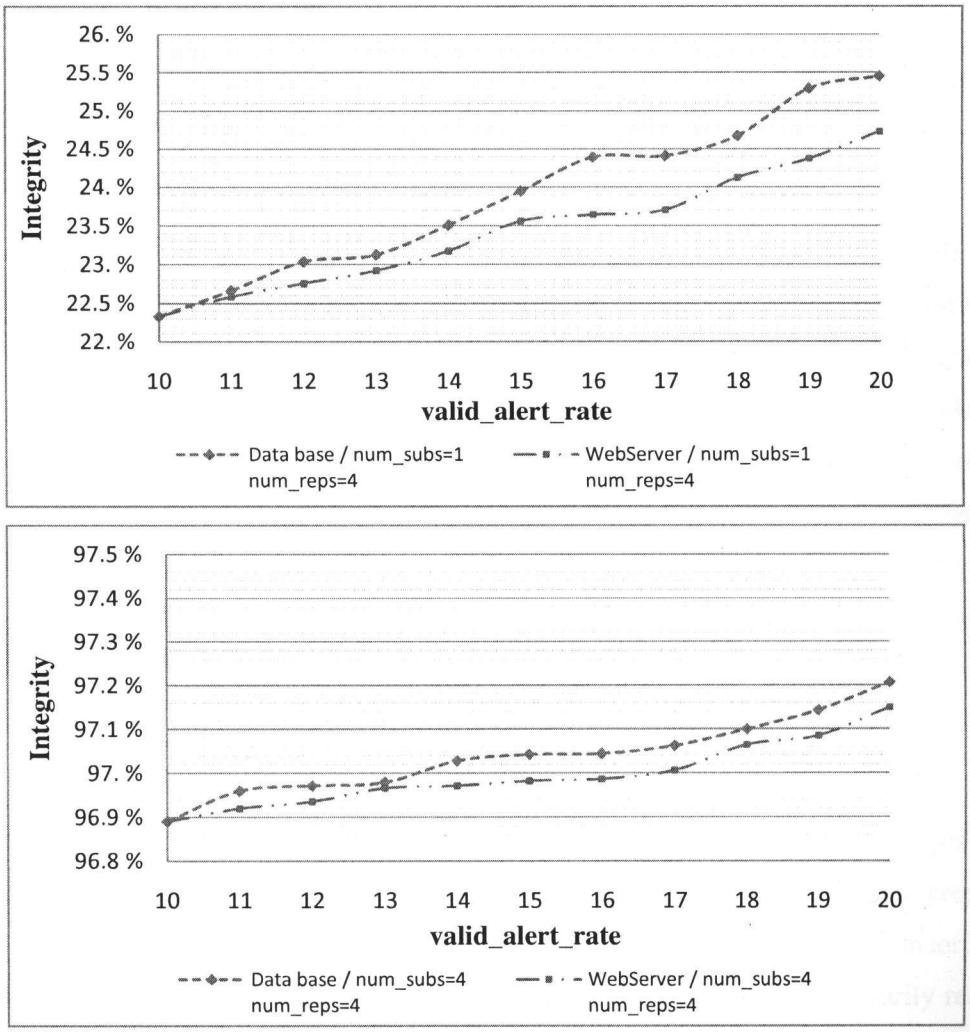


25.1(a)



25.1(b)

Figure 25.1- Comparison of DB replica and WS replica IDSW quality Changes (Availability)



25.2(a)

25.2(b)

Figure 25.2- Comparison of DB replica and WS replica IDSW quality Changes (Integrity)

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The novelty of our proposed approach is the application of Stochastic Activity Networks (SAN) models in capturing the impacts of attack in different layers of a distributed layered system. To the best of our knowledge, no systematic research has ever been conducted to assess security measure of a layered system. Furthermore, constructing a hypothetical model to analyze security properties of a layered system has been more useful than developing an actual system to evaluate the security measures. The proposed SAN models have been designed to be modular models, thus, they can easily be adopted by any other layered intrusion tolerant system. The results obtained from the set of studies conducted in this work are quite significant. These results have revealed the effectiveness of probabilistic modeling method for evaluating a layered architecture intrusion tolerant system. Moreover, the results have demonstrated the utility of SAN models to evaluate security measures.

Although the rate values used in this work are not based on those belonging to a practical system, the obtained results from the experiments provide functional insights to the proposed system architecture. One of the very practical outcomes of these results implies that incorporating more number of replicas required per hosts or many sub-systems will not necessarily result in higher security measure in a system. The results show that at a certain point (i.e. 4 sub-systems and 4 replicas in experiments proposed by this work) security measures will achieve stability, showing minimal variation; adding more hardware to the system would only increase the cost of the system construction and operation while system availability and/or integrity will not be improved.

Moreover, it has been noted that it would be quite essential to determine the desirable quality of the IDSW, especially for the system database layer, which has a considerable impact on the system security properties. However, the justification behind constructing any software will provide a great contribution to selecting an IDSW of a higher quality. The results illustrate that while the repair facilities are kept identical, increasing the quality of IDSW in both layers (i.e. in

our experiments from 5 to 15), will decrease system availability. This behaviour is more significant when the quality of the IDSW for a database improves, while a similar modification will cause an increase in the system integrity measures. The experiments illustrated that while maintaining system availability of 97.73%, the improvement of quality of IDSW in database host resulted in having availability of 97.62%. Similar modification for the web server host resulted in an availability of 97.67%. Considering system integrity, having integrity of 96.89%, increasing IDSW quality from 5 to 15 in database host has resulted in obtaining integrity of 98.93%. Identical improvement made within the web server host has resulted in integrity of 97.88%. Hence, the process of identifying the more critical security measure, either the system availability or integrity, is derived by the design intention of the software.

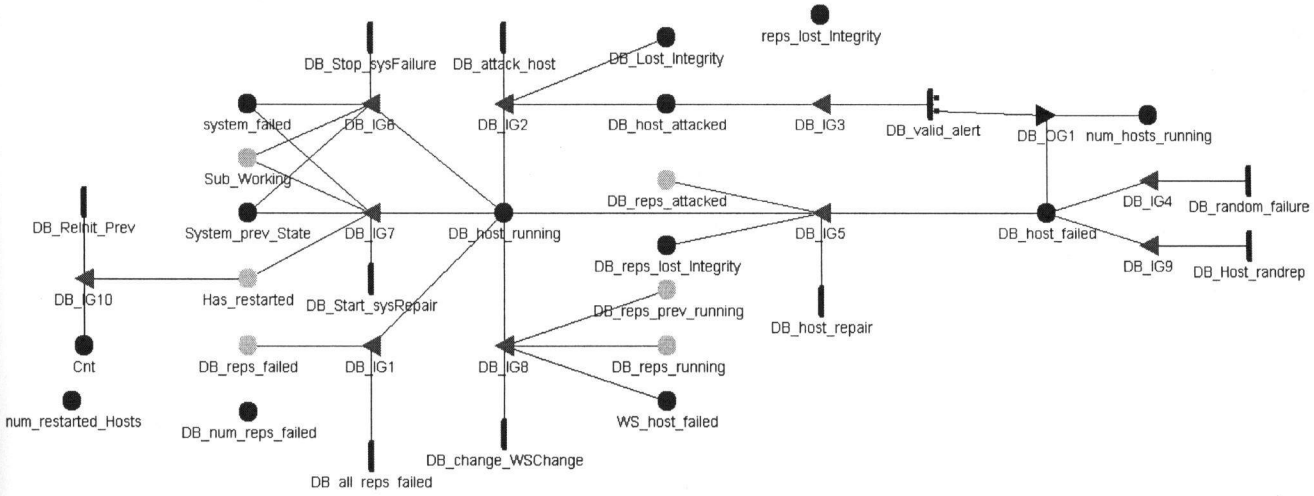
Furthermore, the influence of an intruded lower layer on the security of the system has been analyzed. The results obtained from the experiments indicate that having a system availability of 97.73%, increasing the system host attack rate in the Database layer from 10 to 20 will reduce system availability to 97.55%. Similar modification made to a Web-server layer will contribute to 97.04% availability. This set of results imply that increasing attack rate in Web Server layer has a more severe impact on system availability, while the same modification in Database layer will less severely influence system availability. Similar results have been gathered when measuring integrity of the system under identical set of modification. At system integrity of 96.88%, increasing host attack rate in Database layer has resulted in achieving integrity of 96.68%; similar experiment for Web server layer resulted in system integrity of 96.57%.

The main objective of this study has been to develop a probabilistic assessment method for a three layered intrusion tolerant system – the client layer, the WS layer, and the DB layer. Today's complex systems could consist of multiple layers composing a distributed system. In spite of existing constraints, achieving the goal of the proposed model allows researchers to explore in more depth and detail, the topics on system security, in particular by accounting for the current limitation in the number of layers. In addition, this thesis has considered a one to one relationship between hosts of two layers as well as their replicas. Exploring beyond such relationship between pairs of hosts and replicas (i.e. utilizing many-to-many relationships) will enable future studies to evaluate a multi-layered system with various types of correlation between replicas.

APPENDIX

SYSTEM MODEL

Model: DBHost



Bucket Attributes:	
Place Names	**Initial Markings**
Cnt	0
DB_Lost_Integrity	0
DB_host_attacked	0
DB_host_failed	0
DB_host_running	1
DB_num_reps_failed	0
DB_reps_attacked	0
DB_reps_failed	0
DB_reps_lost_Integrity	0
DB_reps_prev_running	0
DB_reps_running	0
Has_restarted	0
Sub_Working	1
System_prev_State	0
WS_host_failed	0
num_hosts_running	Total_num_hosts
num_restarted_Hosts	0
reps_lost_Integrity	0
system_failed	0
Timed Activity:	**DB_Host_randrep**

Distribution Parameters	**Rate** DB_Host_randrep_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_ReInit_Prev**
Distribution Parameters	**Rate** DB_ReInit_Prev_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_Start_sysRepair**
Distribution Parameters	**Rate** DB_Start_sysRepair_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_Stop_sysFailure**
Distribution Parameters	**Rate** DB_Stop_sysFailure_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_all_reps_failed**
Distribution Parameters	**Rate** DB_all_reps_failed_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_attack_host**
Distribution Parameters	**Rate** DB_attack_host_rate /**((WS_reps_attacked->Mark() & DB_host_WSrep_id->Mark())+1)*/ /*check if the WS replica who called this DB host, is under attack or not*/
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_change_WSChange**
Distribution Parameters	**Rate** DB_change_WSChange_rate
Activation Predicate	(none)

Reactivation Predicate	(none)
Timed Activity:	**DB_host_repair**
Distribution Parameters	**Rate** DB_host_repair_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_random_failure**
Distribution Parameters	**Rate** DB_random_failure_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_valid_alert**
Distribution Parameters	**Rate** DB_valid_alert_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Case Distributions	**case 1** 1- DB_prob_succ **case 2** DB_prob_succ
Input Gate:	**DB_IG1**
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_num_reps_failed->Mark()== Total_num_reps

Function	Sub_Working->DB->Mark()=0;
	DB_host_running->Mark()=0;
	DB_host_failed->Mark()=-1;// APR 06
	num_hosts_running->Mark()--;

Input Gate:	**DB_IG10**
Predicate	num_restarted_Hosts->Mark()==Total_num_hosts && System_prev_State->Mark()==1 && Has_restarted->DB->Mark()==1 && Has_restarted->WS- >Mark()==1
Function	Has_restarted->DB->Mark()=0;

	Has_restarted->WS->Mark()=0; Cnt->Mark()-=2; if (Cnt->Mark()==0) System_prev_State->Mark()=0;
Input Gate:	**DB_IG2**
Predicate	/*Host is not already corrupt and it is running*/ system_failed->Mark()==0 && System_prev_State->Mark()==0 && DB_host_attacked->Mark()==0 && Sub_Working->WS->Mark()==1 && Sub_Working->DB->Mark()==1
Function	DB_host_attacked->Mark()=1; DB_Lost_Integrity->Mark()++;
Input Gate:	**DB_IG3**
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && DB_host_attacked->Mark()==1 && Sub_Working->WS->Mark()==1 && Sub_Working->DB->Mark()==1
Function	;
Input Gate:	**DB_IG4**
Predicate	num_hosts_running->Mark()>0 && Sub_Working->WS->Mark()==1 && Sub_Working->DB->Mark()==1 && DB_host_failed->Mark()==0 && system_failed->Mark()==0 && System_prev_State->Mark()==0
Function	DB_host_failed->Mark()=2; DB_host_running->Mark()=0; /*decreamenting # of running host in the system*/ num_hosts_running->Mark()--; Sub_Working->DB->Mark()=0; for (int i=0;i<Total_num_reps;i++) DB_reps_running->Index(i)->Mark()=0;
Input Gate:	**DB_IG5**
Predicate	num_hosts_running->Mark()>0 && DB_host_failed->Mark()==1 && system_failed->Mark()==0 && System_prev_State->Mark()==0

Function	DB_host_failed->Mark()=0;
	num_hosts_running->Mark()++;
	DB_host_running->Mark()=1;
	DB_host_attacked->Mark()=0;
	Sub_Working->DB->Mark()=1;
	if (reps_lost_Integrity->Mark()>=DB_reps_lost_Integrity-
	>Mark())
	reps_lost_Integrity->Mark()-=DB_reps_lost_Integrity-

	>Mark();
	DB_reps_lost_Integrity->Mark()=0;
	DB_num_reps_failed->Mark()=0;
	for (int i=0;i<Total_num_reps;i++){
	DB_reps_failed->Index(i)->Mark()=0;
	DB_reps_prev_running->Index(i)->Mark()=0;
	DB_reps_running->Index(i)->Mark()=0;
	DB_reps_attacked->Index(i)->Mark()=0;
	}

Input Gate:	DB_IG6
Predicate	system_failed->Mark()==1 && DB_host_running->Mark()!=0
Function	DB_host_running->Mark()=0; for (int i=0;i<Total_num_reps;i++) DB_reps_running->Index(i)->Mark()=0; Sub_Working->DB->Mark()=0; num_hosts_running->Mark()--;

Input Gate:	DB_IG7
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==1 && Has_restarted->DB->Mark()==0

Function	DB_host_running->Mark()=1;
	DB_host_failed->Mark()=0;
	DB_host_attacked->Mark()=0;
	DB_Lost_Integrity->Mark()=0;////////////////// it was commented
	
	DB_reps_lost_Integrity->Mark()=0;
	DB_num_reps_failed->Mark()=0;
	num_restarted_Hosts->Mark()++;
	Cnt->Mark()++;
	if (Sub_Working->DB->Mark()==0)
	num_hosts_running->Mark()++;
	
	Has_restarted->DB->Mark()=1;
	Sub_Working->DB->Mark()=1;
	for (int i=0;i<Total_num_reps;i++){
	DB_reps_failed->Index(i)->Mark()=0;
	DB_reps_running->Index(i)->Mark()=0;
	DB_reps_prev_running->Index(i)->Mark()=0;
	DB_reps_attacked->Index(i)->Mark()=0;
	}

Input Gate:	DB_IG8

Predicate	num_hosts_running->Mark()>0 && system_failed->Mark()==0 && System_prev_State->Mark()==0 && DB_host_failed->Mark() !=1 && //DB_host_failed->Mark() ==0 && DB_host_running->Mark() != Sub_Working->WS->Mark()

Function	if (Sub_Working->WS->Mark()==1){
	int i, n=0;
	for (i=0;i<Total_num_reps;i++)
	if (DB_reps_failed->Index(i)->Mark()==1)
	n++;
	if (n<Total_num_reps){
	DB_host_running->Mark()=1;
	Sub_Working->DB->Mark()=1;
	DB_host_failed->Mark()=0; /*Apr 06*/
	num_hosts_running->Mark()++;
	for (int i=0;i<Total_num_reps;i++)
	DB_reps_running->Index(i)-
	>Mark()=DB_reps_prev_running->Index(i)->Mark();
	}
	}
	/*else if (Sub_Working->WS->Mark()==0 && WS_host_failed-
	>Mark()==1){ Apr 06 */
	else if (Sub_Working->WS->Mark()==0 && WS_host_failed->Mark()!=
	0){
	DB_host_running->Mark()=0;
	Sub_Working->DB->Mark()=0;
	DB_host_failed->Mark()=-1; /*Apr 06*/
	num_hosts_running->Mark()--;
	for (int i=0;i<Total_num_reps;i++){
	DB_reps_prev_running->Index(i)-
	>Mark()=DB_reps_running->Index(i)->Mark();
	DB_reps_running->Index(i)->Mark()=-1;
	}
	}

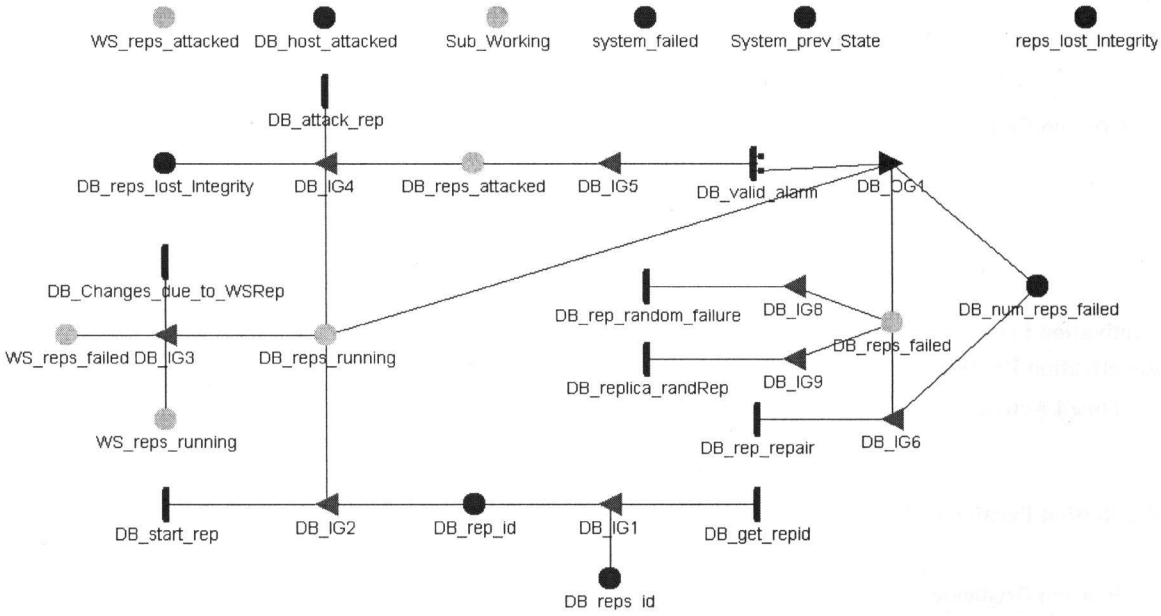
Input Gate:	**DB_IG9**
Predicate	num_hosts_running->Mark()>0 && DB_host_failed->Mark()==2 && system_failed->Mark()==0 && System_prev_State->Mark()==0

Function	DB_host_failed->Mark()=0;
	num_hosts_running->Mark()++;
	DB_host_running->Mark()=1;
	DB_host_attacked->Mark()=0;
	Sub_Working->DB->Mark()=1;
	DB_Lost_Integrity->Mark()=0; /* the only diferences between this
	activity and DB_Host_Repair*/
	DB_reps_lost_Integrity->Mark()=0;/////////////Apr 06
	for (int i=0;i<Total_num_reps;i++){
	DB_reps_failed->Index(i)->Mark()=0;

	DB_reps_prev_running->Index(i)->Mark()=0; DB_reps_running->Index(i)->Mark()=0; DB_reps_attacked->Index(i)->Mark()=0; }

Output Gate:	DB_OG1
Function	DB_host_failed->Mark()=1; DB_host_running->Mark()=0; DB_host_attacked->Mark()=0; DB_Lost_Integrity->Mark()--; num_hosts_running->Mark()--; Sub_Working->DB->Mark()=0; for (int i=0;i<Total_num_reps;i++) DB_reps_running->Index(i)->Mark()=0;

Model: DBReplica



Bucket Attributes:	
Place Names	**Initial Markings**
DB_host_attacked	0
DB_num_reps_failed	0
DB_rep_id	0
DB_reps_attacked	0
DB_reps_failed	0
DB_reps_id	Total_num_reps
DB_reps_lost_Integrity	0
DB_reps_running	0

Sub_Working	1
System_prev_State	0
WS_reps_attacked	0
WS_reps_failed	0
WS_reps_running	0
reps_lost_Integrity	0
system_failed	0

Timed Activity:	**DB_Changes_due_to_WSRep**
Distribution Parameters	**Rate** `DB_Changes_due_to_WSRep_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**DB_attack_rep**

Distribution Parameters	**Rate**
	`DB_base_rep_attack_rate * (DB_host_attacked->Mark()+1.0) * ((WS_reps_attacked->Index(DB_rep_id->Mark()-1)->Mark())+1.0)`
	`/*`
	`WSReplica attacks propagate into DBReplica,`
	`and also DBHost attacks propagate into replicas running on it.`
	`*/`

Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**DB_get_repid**

Distribution Parameters	**Rate**
	`DB_replica_getid_rate`
	`/*WS_replicas_id :common for all reps*/`

Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**DB_rep_random_failure**
Distribution Parameters	**Rate** `DB_rep_random_failure_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**DB_rep_repair**
Distribution Parameters	**Rate** `DB_replica_repair_rate`

Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_replica_randRep**
Distribution Parameters	**Rate** DB_replica_randRep_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_start_rep**
Distribution Parameters	**Rate** DB_start_rep_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**DB_valid_alarm**
Distribution Parameters	**Rate** DB_rep_valid_alarm_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Case Distributions	**case 1** 1 - DB_rep_prob_succ **case 2** DB_rep_prob_succ

Input Gate:	DB_IG1
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && /* the host this replica running on,is not failed*/ DB_reps_id->Mark()>0 && /* there is still replica to be assigned*/ DB_rep_id->Mark()==0 /* there is still replica to be assigned*/
Function	DB_rep_id->Mark()= DB_reps_id->Mark(); DB_reps_id->Mark()--;

Input Gate:	DB_IG2
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_rep_id->Mark()>0 && DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()==0 && (DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()==0 ||

	DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()==-1) //DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()!=1 /* 1. Host submodel will put 1 in this place as needed 2. if DB replica has been previously activated, it forces WS_rep as well to start running and WS_replica will put 1 in its relevant vector of WS_reps_activated */
Function	DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()=1;
Input Gate:	**DB_IG3**
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_rep_id->Mark() >0 && DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()==0 && (WS_reps_running->Index(DB_rep_id->Mark()-1)->Mark() != DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()) /* if the relevant replica in DB group has been activated */

Function	if (WS_reps_running->Index(DB_rep_id->Mark()-1)->Mark()==1 &&
	DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()==0){
	DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()=1;
	}
	if (WS_reps_running->Index(DB_rep_id->Mark()-1)->Mark()==0 &&
	WS_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()==1)
	DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()=0;

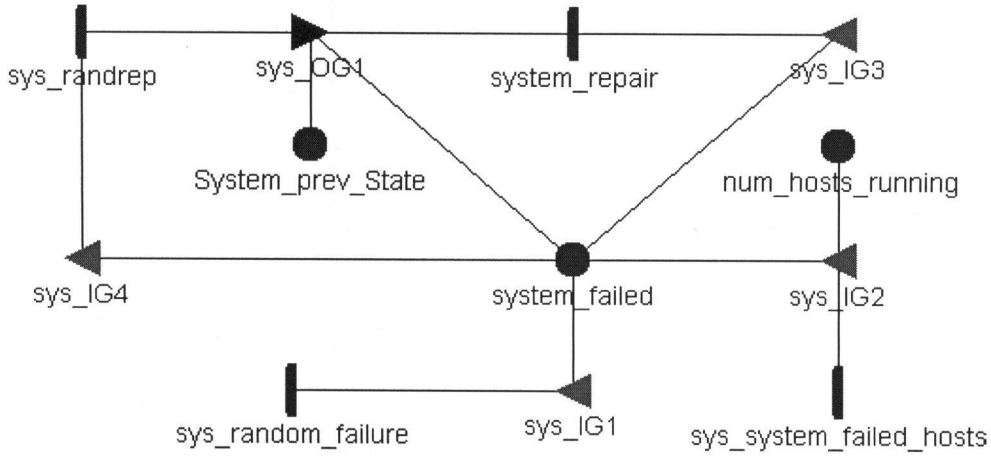
Input Gate:	**DB_IG4**
Predicate	/* Replica is not already under attack and it is running and the host rep running on, is running */ system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_rep_id->Mark()>0 && DB_reps_attacked->Index(DB_rep_id->Mark()-1)->Mark()==0 && DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()==1
Function	DB_reps_attacked->Index(DB_rep_id->Mark()-1)->Mark()=1; /*Shows Replica is under attack*/ DB_reps_lost_Integrity->Mark()++; reps_lost_Integrity->Mark()++;
Input Gate:	**DB_IG5**

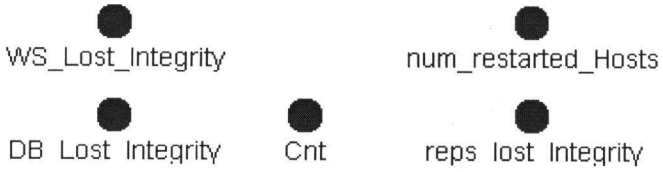
Predicate	`system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_rep_id->Mark()>0 && DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()==0 && DB_reps_attacked->Index(DB_rep_id->Mark()-1)->Mark()==1`
Function	`;`
Input Gate:	**DB_IG6**
Predicate	`/* Replica may be repaired if its failure is due to attack on replica, which if DB_replica_failed==1, otherwise the failure is due to host or WS attack propagation and can not be repaired. */ system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_rep_id->Mark() >0 && DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()==1`
Function	`/* inform the host that this replica has been repaired*/ DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()= 0 ; DB_reps_attacked->Index(DB_rep_id->Mark()-1)->Mark()=0; DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()=1; DB_num_reps_failed->Mark()--;`
Input Gate:	**DB_IG8**
Predicate	`system_failed->Mark()==3 && system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_rep_id->Mark() >0 && DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()==0 && DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()==1`
Function	`DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()=2; DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()=0; DB_num_reps_failed->Mark()++;`
Input Gate:	**DB_IG9**
Predicate	`system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_rep_id->Mark() >0 && DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()==2`
Function	`/* inform the host that this replica has been repaired*/ DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark()= 0 ; DB_reps_attacked->Index(DB_rep_id->Mark()-1)->Mark()=0; DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()=1;`

	DB_reps_lost_Integrity->Mark()=0; DB_num_reps_failed->Mark()--;

Output Gate:	DB_OG1
Function	DB_reps_running->Index(DB_rep_id->Mark()-1)->Mark()=0; /*this Replica is not running any more*/ /* inform the host that this replica has been failed*/ DB_reps_failed->Index(DB_rep_id->Mark()-1)->Mark() = 1; /*Replica doesn't run anymore,so the Integrity is not lost*/ if (DB_reps_lost_Integrity->Mark()>0) DB_reps_lost_Integrity->Mark()--; reps_lost_Integrity->Mark()--; DB_num_reps_failed->Mark()++;

Model: System





Bucket Attributes:

Place Names	Initial Markings
Cnt	0
DB_Lost_Integrity	0
System_prev_State	0
WS_Lost_Integrity	0
num_hosts_running	Total_num_hosts
num_restarted_Hosts	0
reps_lost_Integrity	0

system_failed	0
Timed Activity:	**sys_random_failure**
Distribution Parameters	**Rate** `sys_random_failure_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**sys_randrep**
Distribution Parameters	**Rate** `sys_randrep_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**sys_system_failed_hosts**

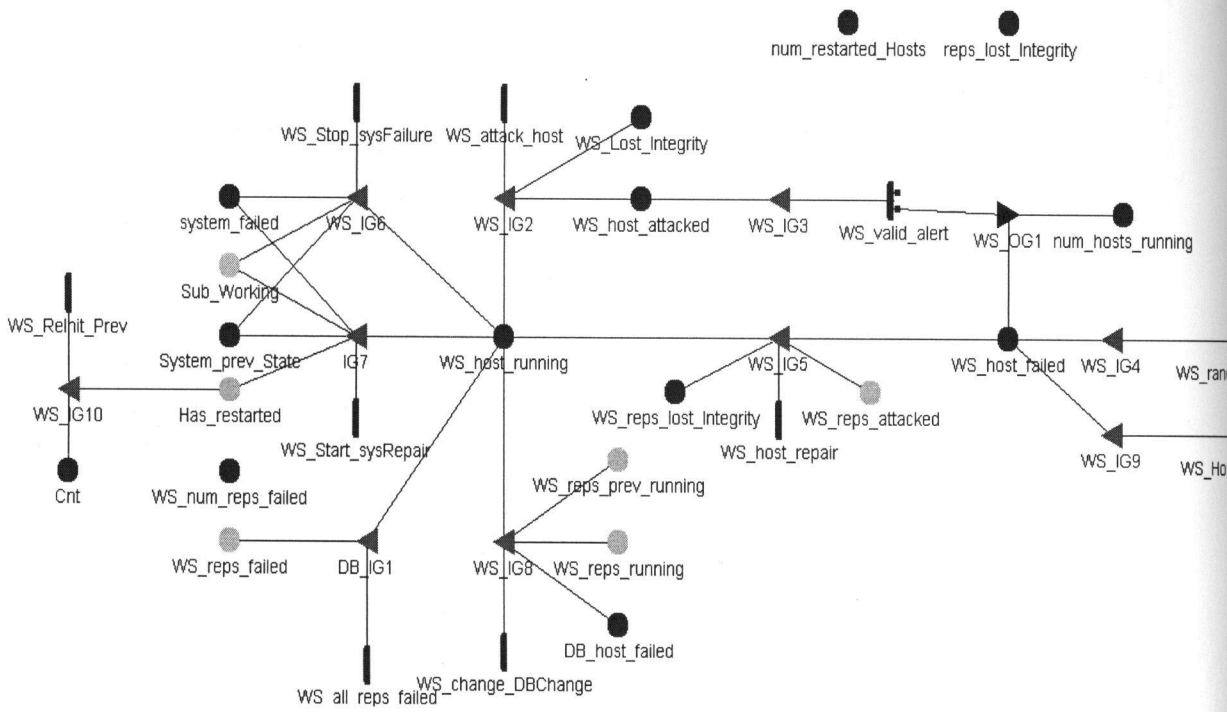
Distribution Parameters	**Rate**
	`SYS_system_failed_hosts_rate`
	`/*DB_host causes system failure */`

Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**system_repair**
Distribution Parameters	**Rate** `system_repair_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Input Gate:	**sys_IG1**
Predicate	`system_failed->Mark()==0`
Function	`system_failed->Mark()=2;` `System_prev_State->Mark()=0;`
Input Gate:	**sys_IG2**
Predicate	`system_failed->Mark()==0 &&` `num_hosts_running->Mark()==0 && System_prev_State->Mark()==0`
Function	`system_failed->Mark()=1;` `System_prev_State->Mark()=0;`
Input Gate:	**sys_IG3**
Predicate	`system_failed->Mark()==1`
Function	`;`
Input Gate:	**sys_IG4**

Predicate	`system_failed->Mark()==2`
Function	`;`
Output Gate:	**sys_OG1**

Function	`System_prev_State->Mark()=1;`
	`system_failed->Mark()=0;`
	`num_restarted_Hosts->Mark()=0;`
	`Cnt->Mark()=0;`
	`WS_Lost_Integrity->Mark()=0;`
	`DB_Lost_Integrity->Mark()=0;`
	`reps_lost_Integrity->Mark()=0;`

Model: WSHost



Bucket Attributes:	
Place Names	**Initial Markings**
Cnt	0
DB_host_failed	0
Has_restarted	0
Sub_Working	1
System_prev_State	0
WS_Lost_Integrity	0
WS_host_attacked	0
WS_host_failed	0

WS_host_running	1
WS_num_reps_failed	0
WS_reps_attacked	0
WS_reps_failed	0
WS_reps_lost_Integrity	0
WS_reps_prev_running	0
WS_reps_running	0
num_hosts_running	Total_num_hosts
num_restarted_Hosts	0
reps_lost_Integrity	0
system_failed	0

Timed Activity:	**WS_Host_randrep**
Distribution Parameters	**Rate** `WS_Host_randrep_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**WS_ReInit_Prev**
Distribution Parameters	**Rate** `WS_ReInit_Prev_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**WS_Start_sysRepair**
Distribution Parameters	**Rate** `WS_Start_sysRepair_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**WS_Stop_sysFailure**
Distribution Parameters	**Rate** `WS_Stop_sysFailure_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**WS_all_reps_failed**
Distribution Parameters	**Rate** `WS_all_reps_failed_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**WS_attack_host**
Distribution Parameters	**Rate** `WS_attack_host_rate`
Activation Predicate	(none)

Reactivation Predicate	(none)
Timed Activity:	**WS_change_DBChange**
Distribution Parameters	**Rate** WS_change_DBChange_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_host_repair**
Distribution Parameters	**Rate** WS_host_repair_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_random_failure**
Distribution Parameters	**Rate** WS_random_failure_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_valid_alert**
Distribution Parameters	**Rate** WS_valid_alert_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Case Distributions	**case 1** 1- WS_prob_succ **case 2** WS_prob_succ

Input Gate:	DB_IG1
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && WS_num_reps_failed->Mark()== Total_num_reps /* system_failed->Mark()==0 && (System_prev_State->Mark()==0 || Sub_restarted->Mark()==1) && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && DB_num_reps_failed->Mark()==Total_num_reps*/

Function	Sub_Working->WS->Mark()=0;
	WS_host_running->Mark()=0;
	WS_host_failed->Mark()=-1;// APR 06
	num_hosts_running->Mark()--;

Input Gate:	**IG7**

Predicate	`system_failed->Mark()==0 &&`
	`System_prev_State->Mark()==1 &&`
	`Has_restarted->WS->Mark()==0`

Function	`WS_host_running->Mark()=1;`
	`WS_host_failed->Mark()=0;`
	`WS_host_attacked->Mark()=0;`
	`/*if(num_restarted_Hosts->Mark()== 0)`
	`WS_Lost_Integrity->Mark()=0;*/`
	`num_restarted_Hosts->Mark()++;`
	`Cnt->Mark()++;`
	`WS_reps_lost_Integrity->Mark()=0;`
	`WS_num_reps_failed->Mark()=0;`
	`if (Sub_Working->WS->Mark()==0)`
	`num_hosts_running->Mark()++;`
	
	`Has_restarted->WS->Mark()=1;`
	
	`/*if(num_restarted_Hosts->Mark()==Total_num_hosts){`
	`System_prev_State->Mark()=0;`
	`Has_restarted->WS->Mark()=0;`
	`Has_restarted->DB->Mark()=0;`
	`}*/`
	
	`Sub_Working->WS->Mark()=1;`
	`for (int i=0;i<Total_num_reps;i++){`
	`WS_reps_failed->Index(i)->Mark()=0;`
	`WS_reps_running->Index(i)->Mark()=0;`
	`WS_reps_prev_running->Index(i)->Mark()=0;`
	`WS_reps_attacked->Index(i)->Mark()=0;`
	`}`

Input Gate:	**WS_IG10**

Predicate	`num_restarted_Hosts->Mark()==Total_num_hosts &&`
	`System_prev_State->Mark()==1 &&`
	`Has_restarted->DB->Mark()==1 && Has_restarted->WS-`
	`>Mark()==1`

Function	`Has_restarted->DB->Mark()=0;`
	`Has_restarted->WS->Mark()=0;`
	`Cnt->Mark()-=2;`
	`if (Cnt->Mark()==0)`
	`System_prev_State->Mark()=0;`

Input Gate:	**WS_IG2**
Predicate	`/* Host is not already corrupt and it is running*/`

	system_failed->Mark()==0 && System_prev_State->Mark()==0 && WS_host_attacked->Mark()==0 && Sub_Working->WS->Mark()==1 && Sub_Working->DB->Mark()==1
Function	WS_host_attacked->Mark()=1; WS_Lost_Integrity->Mark()++;
Input Gate:	**WS_IG3**
Predicate	/* WS_host_attacked->Mark()==1 && WS_host_failed->Mark()==0 && system_failed->Mark()==0 */ WS_host_attacked->Mark()==1 && Sub_Working->WS->Mark()==1 && Sub_Working->DB->Mark()==1 && system_failed->Mark()==0 && System_prev_State->Mark()==0
Function	;
Input Gate:	**WS_IG4**
Predicate	num_hosts_running->Mark()>0 && Sub_Working->WS->Mark()==1 && Sub_Working->DB->Mark()==1 && WS_host_failed->Mark()==0 && system_failed->Mark()==0 && System_prev_State->Mark()==0
Function	WS_host_failed->Mark()=2; WS_host_running->Mark()=0; /*decreamenting # of running host in the system*/ num_hosts_running->Mark()--; Sub_Working->WS->Mark()=0; for (int i=0;i<Total_num_reps;i++) WS_reps_running->Index(i)->Mark()=0;
Input Gate:	**WS_IG5**
Predicate	num_hosts_running->Mark()>0 && WS_host_failed->Mark()==1 && system_failed->Mark()==0 && System_prev_State->Mark()==0

Function	num_hosts_running->Mark()++;
	WS_host_running->Mark()=1;
	WS_host_attacked->Mark()=0;
	WS_host_failed->Mark()=0;
	WS_reps_lost_Integrity->Mark()=0;
	if (reps_lost_Integrity->Mark()>=WS_num_reps_failed->Mark())

```
        reps_lost_Integrity->Mark()-=WS_num_reps_failed-
>Mark();
WS_num_reps_failed->Mark()=0;
Sub_Working->WS->Mark()=1;
for (int i=0;i<Total_num_reps;i++){
        WS_reps_failed->Index(i)->Mark()=0;
        WS_reps_prev_running->Index(i)->Mark()=0;
        WS_reps_running->Index(i)->Mark()=0;
        WS_reps_attacked->Index(i)->Mark()=0;
}
```

Input Gate:	WS_IG6
Predicate	`system_failed->Mark()==1 &&` `WS_host_running->Mark()!=0`

Function	`WS_host_running->Mark()=0;`
	`for (int i=0;i<Total_num_reps;i++)`
	`WS_reps_running->Index(i)->Mark()=0;`
	`Sub_Working->WS->Mark()=0;`
	`num_hosts_running->Mark()--;`

Input Gate:	WS_IG8

Predicate	`/* DB_host_running->Mark()!=WS_host_running->Mark() &&`
	`WS_host_failed->Mark()!=1 */`
	
	`num_hosts_running->Mark()>0 &&`
	`system_failed->Mark()==0 && System_prev_State->Mark()==0 &&`
	`WS_host_failed->Mark()!=1 &&`
	`WS_host_running->Mark() != Sub_Working->DB->Mark()`

Function	`if (Sub_Working->DB->Mark()==1){`
	`int i,n;`
	`n=0;`
	`for (i=0;i<Total_num_reps;i++)`
	`if (WS_reps_failed->Index(i)->Mark()==1)`
	`n++;`
	`if (n<Total_num_reps){`
	`WS_host_running->Mark()=1;`
	`Sub_Working->WS->Mark()=1;`
	`WS_host_failed->Mark()=0; /*Apr 06*/`
	`num_hosts_running->Mark()++;`
	`for (int i=0;i<Total_num_reps;i++)`
	`WS_reps_running->Index(i)-`
	`>Mark()=WS_reps_prev_running->Index(i)->Mark();`
	`}`
	`}`
	`else if (Sub_Working->DB->Mark()==0 && DB_host_failed->Mark() !=`
	`0){`
	`WS_host_running->Mark()=0;`
	`Sub_Working->WS->Mark()=0;`
	`WS_host_failed->Mark()=-1; /*Apr 06*/`

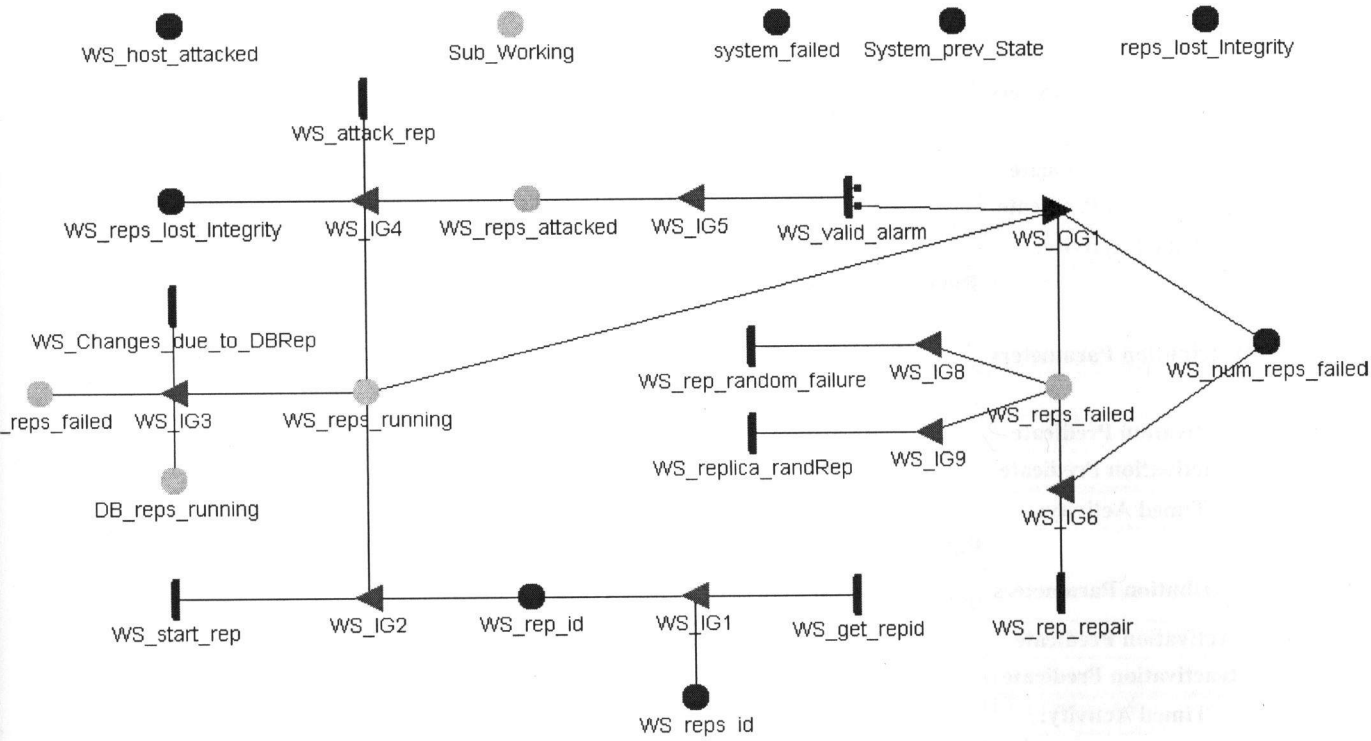
```
        num_hosts_running->Mark()--;
        for (int i=0;i<Total_num_reps;i++){
                WS_reps_prev_running->Index(i)-
>Mark()=WS_reps_running->Index(i)->Mark();
                WS_reps_running->Index(i)->Mark()=-1;
        }
}
```

Input Gate:	WS_IG9
Predicate	num_hosts_running->Mark()>0 && DB_host_failed->Mark()==2 && system_failed->Mark()==0 && System_prev_State->Mark()==0

Function	WS_host_failed->Mark()=0;
	num_hosts_running->Mark()++;
	WS_host_running->Mark()=1;
	WS_host_attacked->Mark()=0;
	Sub_Working->WS->Mark()=1;
	WS_Lost_Integrity->Mark()=0; /* the only diferences between this activity and DB_Host_Repair*/
	WS_reps_lost_Integrity->Mark()=0;////////////Apr 06
	for (int i=0;i<Total_num_reps;i++){
	WS_reps_failed->Index(i)->Mark()=0;
	WS_reps_prev_running->Index(i)->Mark()=0;
	WS_reps_running->Index(i)->Mark()=0;
	WS_reps_attacked->Index(i)->Mark()=0;
	}

Output Gate:	WS_OG1
Function	WS_host_failed->Mark()=1; WS_host_running->Mark()=0; WS_host_attacked->Mark()=0; /*Host doesn't run anymore,so the Integrity is not lost*/ WS_Lost_Integrity->Mark()--; /*decreamenting # of running host in the system*/ num_hosts_running->Mark()--; Sub_Working->WS->Mark()=0; for (int i=0;i<Total_num_reps;i++) WS_reps_running->Index(i)->Mark()=0;

Model: WSReplica



Bucket Attributes:	
Place Names	**Initial Markings**
DB_reps_failed	0
DB_reps_running	0
Sub_Working	1
System_prev_State	0
WS_host_attacked	0
WS_num_reps_failed	0
WS_rep_id	0
WS_reps_attacked	0
WS_reps_failed	0
WS_reps_id	Total_num_reps
WS_reps_lost_Integrity	0
WS_reps_running	0
reps_lost_Integrity	0
system_failed	0

Timed Activity:	**WS_Changes_due_to_DBRep**
Distribution Parameters	**Rate** `WS_Changes_due_to_DBRep_rate`
Activation Predicate	(none)

Reactivation Predicate	(none)
Timed Activity:	**WS_attack_rep**

Distribution Parameters	**Rate**
	`WS_base_rep_attack_rate`
	`* (WS_host_attacked->Mark()+1.0)`

Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_get_repid**

Distribution Parameters	**Rate**
	`WS_replica_getid_rate`
	`/*WS_replicas_id :common for all reps*/`

Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_rep_random_failure**
Distribution Parameters	**Rate** `WS_rep_random_failure_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_rep_repair**
Distribution Parameters	**Rate** `WS_replica_repair_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_replica_randRep**
Distribution Parameters	**Rate** `WS_replica_randRep_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_start_rep**
Distribution Parameters	**Rate** `WS_start_rep_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**WS_valid_alarm**
Distribution Parameters	**Rate** `WS_rep_valid_alarm_rate`
Activation Predicate	(none)

Reactivation Predicate	(none)
Case Distributions	**case 1** 1 - WS_rep_prob_succ **case 2** WS_rep_prob_succ

Input Gate:	WS_IG1
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && /* the host this replica running on,is not failed*/ WS_reps_id->Mark()>0 && /* there is still replica to be assigned*/ WS_rep_id->Mark()==0 /* this replica doesn't have nay id yet*/
Function	WS_rep_id->Mark()= WS_reps_id->Mark(); WS_reps_id->Mark()--;

Input Gate:	WS_IG2
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && WS_rep_id->Mark()>0 && WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()==0&& WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()!=1 /* 1. Host submodel will put 1 in this place as needed 2. if DB replica has been previously activated, it forces WS_rep as well to start running and WS_replica will put 1 in its relevant vector of WS_reps_activated */
Function	WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()=1;

Input Gate:	WS_IG3
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && WS_rep_id->Mark() >0 && WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()==0 && (WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark() != DB_reps_running->Index(WS_rep_id->Mark()-1)->Mark()) /* if the relevant replica in DB group has been activated */

Function	`if (DB_reps_running->Index(WS_rep_id->Mark()-1)->Mark()==1 &&`
	`  WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()==0){`
	`  WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()=1;`
	`}`
	`if (DB_reps_running->Index(WS_rep_id->Mark()-1)->Mark()==0 &&`
	`  DB_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()==1)`
	`  WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()=0;`

Input Gate:	**WS_IG4**

Predicate	`/*`
	`  Replica is not already under attack and it is running`
	`*/`
	`system_failed->Mark()==0  && System_prev_State->Mark()==0 &&`
	`Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 &&`
	`WS_rep_id->Mark()>0 &&`
	`WS_reps_attacked->Index(WS_rep_id->Mark()-1)->Mark()==0 &&`
	`WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()==1`

Function	`WS_reps_attacked->Index(WS_rep_id->Mark()-1)->Mark()=1;`
	`/* to keep track of all the reps which have been attacked since presence`
	`  of attack in rep increases the vulnerability of DB sobmodel, we need to know`
	`  which WS replicas have been attacked.*/`
	`WS_reps_lost_Integrity->Mark()++;`
	`reps_lost_Integrity->Mark()++;`

Input Gate:	**WS_IG5**

Predicate	`system_failed->Mark()==0 && System_prev_State->Mark()==0 &&`
	`Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 &&`
	`WS_rep_id->Mark()>0 &&`
	`WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()==0`
	`&& WS_reps_attacked->Index(WS_rep_id->Mark()-1)->Mark()==1`

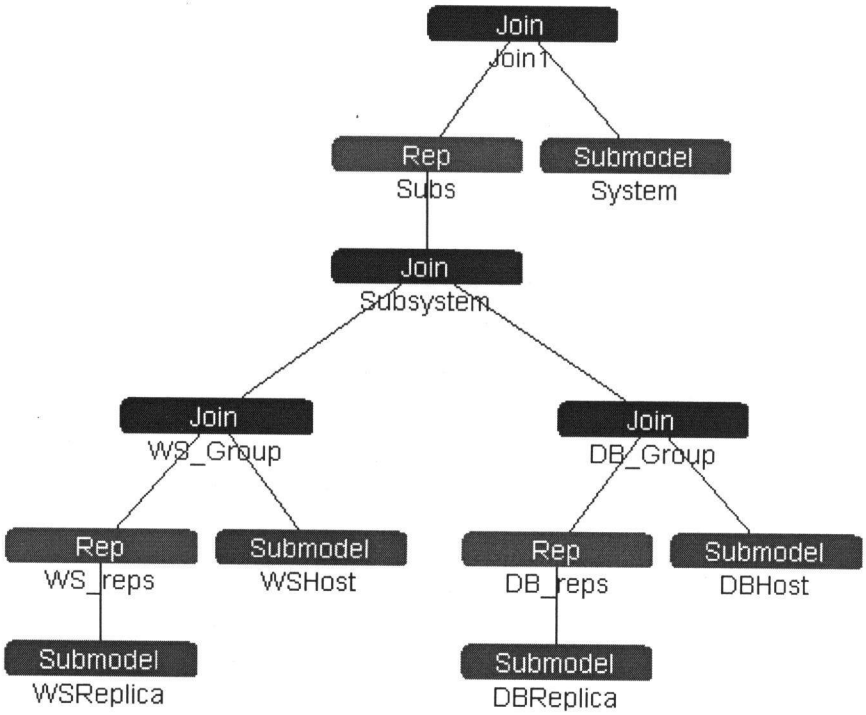
Function	`;`
Input Gate:	**WS_IG6**

Predicate	`/* Replica may be repaired if its failure is due to attack on replica, which`
	`  if WS_replica_failed==1,otherwise the failure is due to host attack propagation`
	`  and can not be repaired.`
	`*/`
	`system_failed->Mark()==0 && System_prev_State->Mark()==0 &&`
	`Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 &&`
	`WS_rep_id->Mark() >0 &&`
	`WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()==1`

Function	

	/* inform the host that this replica has been repaired*/ WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()= 0 ; WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()=1; WS_num_reps_failed->Mark()--;
Input Gate:	**WS_IG8**
Predicate	system_failed->Mark()==3 && system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && WS_rep_id->Mark() >0 && WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()==0 && WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()==1
Function	WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()=2; WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()=0; WS_num_reps_failed->Mark()++;
Input Gate:	**WS_IG9**
Predicate	system_failed->Mark()==0 && System_prev_State->Mark()==0 && Sub_Working->DB->Mark()==1 && Sub_Working->WS->Mark()==1 && WS_rep_id->Mark() >0 && WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()==2
Function	/* inform the host that this replica has been repaired*/ WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark()= 0 ; WS_reps_attacked->Index(WS_rep_id->Mark()-1)->Mark()=0; WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()=1; WS_reps_lost_Integrity->Mark()=0; WS_num_reps_failed->Mark()--;
Output Gate:	**WS_OG1**
Function	/*this Replica is not running any more*/ WS_reps_running->Index(WS_rep_id->Mark()-1)->Mark()=0; /* inform the host that this replica has been failed*/ WS_reps_failed->Index(WS_rep_id->Mark()-1)->Mark() = 1; /*Replica doesn't run anymore,so the Integrity is not lost*/ WS_reps_attacked->Index(WS_rep_id->Mark()-1)->Mark()=0; /*Reinitializing marking of the places*/ WS_reps_lost_Integrity->Mark()--; if (reps_lost_Integrity->Mark()>0)reps_lost_Integrity->Mark()--; WS_num_reps_failed->Mark()++;

Model: System_Architecture



Rep Node	Reps	Shared State Variables
DB_reps	Total_num_reps	DB_host_attacked
		DB_num_reps_failed
		DB_reps_attacked
		DB_reps_failed
		DB_reps_id
		DB_reps_lost_Integrity
		DB_reps_running
		Sub_Working
		System_prev_State
		WS_reps_attacked
		WS_reps_failed
		WS_reps_running
		reps_lost_Integrity
		system_failed
Subs	Total_num_subs	Cnt
		DB_Lost_Integrity
		System_prev_State
		WS_Lost_Integrity
		num_hosts_running
		num_restarted_Hosts

		reps_lost_Integrity
		system_failed
WS_reps	Total_num_reps	DB_reps_failed
		DB_reps_running
		Sub_Working
		System_prev_State
		WS_host_attacked
		WS_num_reps_failed
		WS_reps_attacked
		WS_reps_failed
		WS_reps_id
		WS_reps_lost_Integrity
		WS_reps_running
		reps_lost_Integrity
		system_failed

Join Node: DB_Group :

State Variable Name	Submodel Variables
Cnt	DBHost->Cnt
DB_Lost_Integrity	DBHost->DB_Lost_Integrity
DB_host_attacked	DBHost->DB_host_attacked
	DB_reps->DB_host_attacked
DB_host_failed	DBHost->DB_host_failed
DB_host_running	DBHost->DB_host_running
DB_num_reps_failed	DB_reps->DB_num_reps_failed
	DBHost->DB_num_reps_failed
DB_reps_attacked	DB_reps->DB_reps_attacked
	DBHost->DB_reps_attacked
DB_reps_failed	DBHost->DB_reps_failed
	DB_reps->DB_reps_failed
DB_reps_id	DB_reps->DB_reps_id
DB_reps_lost_Integrity	DB_reps->DB_reps_lost_Integrity
	DBHost->DB_reps_lost_Integrity
DB_reps_running	DBHost->DB_reps_running
	DB_reps->DB_reps_running
Has_restarted	DBHost->Has_restarted
Sub_Working	DBHost->Sub_Working
	DB_reps->Sub_Working
System_prev_State	DB_reps->System_prev_State
	DBHost->System_prev_State
WS_host_failed	DBHost->WS_host_failed
WS_reps_attacked	DB_reps->WS_reps_attacked
WS_reps_failed	DB_reps->WS_reps_failed
WS_reps_running	DB_reps->WS_reps_running

num_hosts_running	DBHost->num_hosts_running
num_restarted_Hosts	DBHost->num_restarted_Hosts
reps_lost_Integrity	DB_reps->reps_lost_Integrity
	DBHost->reps_lost_Integrity
system_failed	DB_reps->system_failed
	DBHost->system_failed

Join Node: Join1 :

State Variable Name	Submodel Variables
Cnt	Subs->Cnt
	System->Cnt
DB_Lost_Integrity	Subs->DB_Lost_Integrity
	System->DB_Lost_Integrity
System_prev_State	System->System_prev_State
	Subs->System_prev_State
WS_Lost_Integrity	Subs->WS_Lost_Integrity
	System->WS_Lost_Integrity
num_hosts_running	System->num_hosts_running
	Subs->num_hosts_running
num_restarted_Hosts	Subs->num_restarted_Hosts
	System->num_restarted_Hosts
reps_lost_Integrity	Subs->reps_lost_Integrity
	System->reps_lost_Integrity
system_failed	System->system_failed
	Subs->system_failed

Join Node: Subsystem :

State Variable Name	Submodel Variables
Cnt	WS_Group->Cnt
	DB_Group->Cnt
DB_Lost_Integrity	DB_Group->DB_Lost_Integrity
DB_host_failed	WS_Group->DB_host_failed
	DB_Group->DB_host_failed
DB_reps_failed	WS_Group->DB_reps_failed
	DB_Group->DB_reps_failed
DB_reps_running	WS_Group->DB_reps_running
	DB_Group->DB_reps_running
Has_restarted	WS_Group->Has_restarted
	DB_Group->Has_restarted
Sub_Working	WS_Group->Sub_Working
	DB_Group->Sub_Working
System_prev_State	WS_Group->System_prev_State
	DB_Group->System_prev_State
WS_Lost_Integrity	WS_Group->WS_Lost_Integrity

WS_host_failed	WS_Group->WS_host_failed
	DB_Group->WS_host_failed
WS_reps_attacked	WS_Group->WS_reps_attacked
	DB_Group->WS_reps_attacked
WS_reps_failed	WS_Group->WS_reps_failed
	DB_Group->WS_reps_failed
WS_reps_running	WS_Group->WS_reps_running
	DB_Group->WS_reps_running
num_hosts_running	WS_Group->num_hosts_running
	DB_Group->num_hosts_running
num_restarted_Hosts	WS_Group->num_restarted_Hosts
	DB_Group->num_restarted_Hosts
reps_lost_Integrity	WS_Group->reps_lost_Integrity
	DB_Group->reps_lost_Integrity
system_failed	WS_Group->system_failed
	DB_Group->system_failed

Join Node: WS_Group :

State Variable Name	Submodel Variables
Cnt	WSHost->Cnt
DB_host_failed	WSHost->DB_host_failed
DB_reps_failed	WS_reps->DB_reps_failed
DB_reps_running	WS_reps->DB_reps_running
Has_restarted	WSHost->Has_restarted
Sub_Working	WSHost->Sub_Working
	WS_reps->Sub_Working
System_prev_State	WS_reps->System_prev_State
	WSHost->System_prev_State
WS_Lost_Integrity	WSHost->WS_Lost_Integrity
WS_host_attacked	WSHost->WS_host_attacked
	WS_reps->WS_host_attacked
WS_host_failed	WSHost->WS_host_failed
WS_num_reps_failed	WS_reps->WS_num_reps_failed
	WSHost->WS_num_reps_failed
WS_reps_attacked	WS_reps->WS_reps_attacked
	WSHost->WS_reps_attacked
WS_reps_failed	WSHost->WS_reps_failed
	WS_reps->WS_reps_failed
WS_reps_lost_Integrity	WS_reps->WS_reps_lost_Integrity
	WSHost->WS_reps_lost_Integrity
WS_reps_running	WSHost->WS_reps_running
	WS_reps->WS_reps_running
num_hosts_running	WSHost->num_hosts_running
num_restarted_Hosts	WSHost->num_restarted_Hosts

reps_lost_Integrity	WS_reps->reps_lost_Integrity
	WSHost->reps_lost_Integrity
system_failed	WS_reps->system_failed
	WSHost->system_failed

Performance Variable Model: RW		
Top Level Model Information	Child Model Name	System_Architecture
	Model Type	Rep/Join

Performance Variable : Availability			
Affecting Models	System		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if (System->num_hosts_running->Mark()==0) return 0.0;`
	`if (System->system_failed->Mark() != 0) return 0.0;`
	`return 1.0;`		

Simulator Statistics	Type	Interval of Time	
	Options	Estimate Mean	
		Include Lower Bound on Interval Estimate	
		Include Upper Bound on Interval Estimate	
		Estimate out of Range Probabilities	
		Confidence Level is Relative	
	Parameters	Start Time	0.0,
		Stop Time	100,
	Confidence	Confidence Level	0.99
		Confidence Interval	0.01

Performance Variable : Integrity		
Affecting Models	System	
Impulse Functions		

Reward Function	*(Reward is over all Available Models)*
	`if ((System->WS_Lost_Integrity->Mark() + System->DB_Lost_Integrity->Mark()) > ((Total_num_subs * 2.0)/3.0) )`
	`return 0.0 ;`
	`if ( System->reps_lost_Integrity->Mark() > (Total_num_reps * 2.0)/ 3.0)`
	`return 0.0;`
	`return 1.0 ;`	

Simulator Statistics	Type	Interval of Time
	Options	Estimate Mean
		Include Lower Bound on Interval Estimate
		Include Upper Bound on Interval Estimate
		Estimate out of Range Probabilities

		Confidence Level is Relative	
	Parameters	Start Time	0.0,
		Stop Time	100,
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Range Study Variable Assignments for Study *Std_base* in Project *Latest* :

Variable	Type	Range Type	Range	Increment	Increment Type	Function	n
DB_Changes_due_to_WSRep_rate	double	Fixed	10.0	-	-	-	-
DB_Host_randrep_rate	double	Fixed	0	-	-	-	-
DB_ReInit_Prev_rate	double	Fixed	5.0	-	-	-	-
DB_Start_sysRepair_rate	double	Fixed	5.0	-	-	-	-
DB_Stop_sysFailure_rate	double	Fixed	5.0	-	-	-	-
DB_all_reps_failed_rate	double	Fixed	5.0	-	-	-	-
DB_attack_host_rate	double	Fixed	10.0	-	-	-	-
DB_base_rep_attack_rate	double	Fixed	40.0	-	-	-	-
DB_change_WSChange_rate	double	Fixed	10.0	-	-	-	-
DB_host_repair_rate	double	Fixed	5.0	-	-	-	-
DB_prob_succ	double	Fixed	0.99	-	-	-	-
DB_random_failure_rate	double	Fixed	0	-	-	-	-
DB_rep_prob_succ	double	Fixed	0.9	-	-	-	-
DB_rep_random_failure_rate	double	Fixed	0	-	-	-	-
DB_rep_valid_alarm_rate	double	Fixed	10.0	-	-	-	-
DB_replica_getid_rate	double	Fixed	100.0	-	-	-	-
DB_replica_randRep_rate	double	Fixed	0	-	-	-	-
DB_replica_repair_rate	double	Fixed	10.0	-	-	-	-
DB_start_rep_rate	short	Fixed	50	-	-	-	-
DB_valid_alert_rate	double	Fixed	5.0	-	-	-	-
SYS_system_failed_hosts_rate	double	Fixed	1.0	-	-	-	-
Total_num_hosts	double	Fixed	1.0	-	-	-	-
Total_num_reps	double	Incremental	[Incremental Range,1.0]	1.0	Additive	-	-
Total_num_subs	double	Incremental	[Incremental Range,1.0]	1.0	Additive	-	-
WS_Changes_due_to_DBRep_rate	double	Fixed	10.0	-	-	-	-
WS_Host_randrep_rate	double	Fixed	0	-	-	-	-
WS_ReInit_Prev_rate	double	Fixed	5.0	-	-	-	-
WS_Start_sysRepair_rate	double	Fixed	5.0	-	-	-	-
WS_Stop_sysFailure_rate	double	Fixed	5.0	-	-	-	-
WS_all_reps_failed_rate	double	Fixed	5.0	-	-	-	-
WS_attack_host_rate	double	Fixed	10.0	-	-	-	-
WS_base_rep_attack_rate	double	Fixed	40.0	-	-	-	-
WS_change_DBChange_rate	double	Fixed	10.0	-	-	-	-
WS_host_repair_rate	double	Fixed	5.0	-	-	-	-

WS_prob_succ	double	Fixed	0.99	-	-	-	-
WS_random_failure_rate	double	Fixed	0.0	-	-	-	-
WS_rep_prob_succ	double	Fixed	0.9	-	-	-	-
WS_rep_random_failure_rate	double	Fixed	0	-	-	-	-
WS_rep_valid_alarm_rate	double	Fixed	10.0	-	-	-	-
WS_replica_getid_rate	double	Fixed	100.0	-	-	-	-
WS_replica_randRep_rate	double	Fixed	0	-	-	-	-
WS_replica_repair_rate	double	Fixed	10.0	-	-	-	-
WS_start_rep_rate	double	Fixed	50.0	-	-	-	-
WS_valid_alert_rate	double	Fixed	5.0	-	-	-	-
sys_random_failure_rate	double	Fixed	0	-	-	-	-
sys_randrep_rate	double	Fixed	0	-	-	-	-
system_repair_rate	double	Fixed	1.0	-	-	-	-

REFERENCES

[1] D.M. Nicol, W.H. Sanders, and K.S. Trivedi, "Model-based evaluation: From dependability to security", IEEE Trans. on Dependable and Secure Computing, pp. 48-65, Jan.-Mar. 2004.

[2] B. B. Madan, K. Goseva-Popstojanova, K. Vaidyanathan, and K. S. Trivedi, "A method for modeling and quantifying the security attributes of intrusion tolerant systems", In Performance Evaluation, volume 56, 2004.

[3] J. P. Anderson, "Computer Security Technology Planning Study", Technical Report ESD-TR-73-51, vols. I and II, AD-758 206, USAF Electronic Systems Division, October 1972.

[4] R.V. Belani, S.M. Das, and D. Fisher, "One-to-one Modeling and Simulation of Unbounded Systems: Experiences and Lessons", In Proc. of the 2002 Winter Simulation Conference, pp. 720–724, December 2002.

[5] S. Singh, M. Cukier, and W. H. Sanders, "Probabilistic Validation of an Intrusion-Tolerant Replication Systems", In Proc. of Int'l Conf. Dependable Systems and Networks (DSN2003), pp. 616-624, June 2003.

[6] J.C. Laprie, "Dependability of computer systems: concepts, limits, improvements", In Proc. of the ISSRE-95, pp. 2–11, 1995.

[7] K.S. Trivedi, "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", second ed., 2001.

[8] M.L. Shooman, "Probabilistic Reliability: An Engineering Approach", second ed. Malabar, Fla.: R.E. Krieger Publishing Co., 1990.

[9] R.A. Sahner, K.S. Trivedi, and A. Puliafito, "Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package", Kluwer Academic Publishers,1996.

[10] http://www.relexsoftware.com/products/relanalysissoft.asp, 2004.

[11] J.E. Arsenault, and J.A. Roberts, "Reliability and Maintainability of Electronic Systems", Rockville, MD: Computer Science Press, 1980.

[12] R.E. Barlow, and F. Proschan, "Statistical Theory of Reliability and Life Testing", New York: Holt, Rinehart and Winston, 1975.

[13] B.S. Dhillon, and C. Singh, "Engineering Reliability: New Techniques and Applications", New York: Wiley, 1981.

[14] E. Henley, and H. Kumamoto, "Reliability Engineering and Risk Assessment", Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[15] M.V. Higuero, J. J. Unzilla, P. Sáiz, E. Jacob, M. Aguado, and I. Goirizelaia , "A practical tool for analysis of security in systems for distribution of digital contents based on 'attack trees", IEEE Symp. on Broadband Multimedia Systems and Broadcasting, 2009, Spain.

[16] Khand, P.A, "System level Security modeling using Attack trees", 2nd Int'l Conf. on Computer, Control and Communication, 2009.

[17] Vu, H.L. Khaw, K.K. Chen, and T. Fei-Ching Kuo, "A New Approach for Network Vulnerability Analysis", 33rd IEEE Conf. on Local Computer Networks,2008.

[18] Jing-Song Cui, and Da Zhang, "The Research and Application of Security Requirements Analysis Methodology of Information Systems", 2nd Int'l Conf. on Anti-counterfeiting, Security and Identification, 2008.

[19] F. Besson, J. Jensen, D.L. Me´tayer, and T. Thorn, "Model Checking Security Properties of Control Flow Graphs", J. Computer Security, vol. 9, no. 3, pp. 217-250, 2001.

[20] H. Chen, D. Dean, and D. Wagner, "Model Checking One Million Lines of C Code", Proc. 11th Ann. Network and Distributed System Security Symp., 2004.

[21] R.W. Ritchey, and P. Ammann, "Using Model Checking to Analyze Network Vulnerabilities", Proc. IEEE Symp. Security and Privacy, pp. 156-165, May 2000.

[22] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing, "Automated Generation and Analysis of Attack Graphs", Proc. 2002 IEEE Symp. Security and Privacy, pp. 273-284, May 2002.

[23] B. Haverkort, R. Marie, G. Rubino, and K.S. Trivedi, "Performability Modeling Tools and Techniques", Chichester, England: John Wiley & Sons, 2001.

[24] K.S. Trivedi, "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", second ed. New York: John Wiley and Sons, 2001.

[25] G. Bolch, S. Greiner, H. de Meer, and K.S. Trivedi, "Queueing Networks and Markov Chains", New York: John Wiley & Sons, 1998.

[26] R.A. Sahner, K.S. Trivedi, and A. Puliafito, "Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package", Kluwer Academic Publishers, 1996.

[27] H. Hermanns, "Interactive Markov Chains and the Quest for Quantified Quality", Springer, LNCS vol. 2428, 2002.

[28] D. Daly, P. Buchholz, and W.H. Sanders, "An Approach for Bounding Reward Measures in Markov Models Using Aggregation" Technical Report uiul-eng-04-2206 (crhc-04-06), Univ. of Illinois at Urbana-Champaign Coordinated Science Laboratory, July 2004

[29] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper, "Complexity of Memory-Efficient Kronecker Operations with Applications to the Solution of Markov Models", informs J. Computing, vol. 12, no. 3, pp. 203-222, 2000.

[30] A.S. Miner, "Efficient Solution of GSPNs Using Canonical Matrix Diagrams", Proc. Ninth Int'l Workshop Petri Nets and Performance Models, pp. 101-110, Sept. 2001.

[31] G. Ciardo, R.A. Marie, B. Sericola, and K.S. Trivedi, "Performability Analysis Using Semi-Markov Reward Processes", IEEE Trans. Computers, vol. 39, no. 10, pp. 1251-1264, Oct. 1990.

[32] V. Kulkarni, " Modeling and Analysis of Stochastic Systems" New York: Chapman Hall, 1995.

[33] H. Hermanns, "Interactive Markov Chains and the Quest for Quantified Quality", Springer, LNCS vol. 2428, 2002.

[34] M. Bernardo, and R. Gorrieri, "A Tutorial on EMPA: A Theory of Concurrent Processes with Non determinism, Priorities, Probabilities and Time", Theoretical Computer Science, vol. 202, pp. 1-54, 1998.

[35] P. Buchholz, "Markovian Process Algebra: Composition and Equivalence", Proc. Second Workshop Process Algebras and Performance Modelling, Arbeitsberichte des IMMD, vol. 27, no. 4, pp. 11- 30, 1994.

[36] iu Mixia Zhang Qiuyu Yu Dongmei Zhao Hong, "Formal Security Model Research Based on Petri-net" , Granular Computing, 2005 IEEE Int'l Conf., vol. 2, 25-27, pp. 575 – 578, July 2005.

[37] M. Malhotra and K. Trivedi, "Dependability Modeling Using Petri Nets", IEEE Trans. Reliability, vol. 44, no. 3, pp. 428-440, Sept. 1995.

[38] K.S. Trivedi, "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", second ed. New York: John Wiley and Sons, 2001.

[39] M. Ajmone Marsan, G. Balbo, and G. Conte, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems", ACM Trans. Computer Systems, vol. 2, pp. 93-122, 1984.

[40] G. Ciardo, J. Muppala, and K. Trivedi, "SPNP: Stochastic Petri Net Package", Proc. Third Int'l Workshop Petri Nets and Performance Models, pp. 142-151, 1989.

[41] J.B. Dugan, V. Nicola, R. Geist, and K. Trivedi, "Extended Stochastic Petri Nets: Applications and Analysis", Proc. Conf. Performance '84, pp. 507-519, 1985.

[42] J.F. Meyer, A. Movaghar, and W.H. Sanders, "Stochastic Activity Networks: Structure, Behaviour, and Application", Proc. Int'l Workshop Timed Petri Nets, pp. 106-115, July 1985.

[43] W.H. Sanders, and J.F. Meyer, "Stochastic Activity Networks: Formal Definitions and Concepts" Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science, LNCS, no. 2090, pp. 315-343, 2001.

[44] D. L. Parnas, "The influence of software structure on reliability", In Proc. 1975 Int'l Conf. Reliable Software, pp. 358–362, Los Angeles, April 1975.

[45] M. L. Shooman, "Structural models for software reliability prediction". In Proc. 2nd Int'l Conf. Software Engineering, pp. 268–280, October 1976.

[46] D. Hamlet, "Are we testing for true reliability?", IEEE Software, vol. 9, pp. 21–27, July 1992.

[47] W. Farr. Handbook of Software Reliability Engineering, M. R. Lyu, Editor, chapter "Software Reliability Modeling Survey", pp. 71–117. McGraw-Hill, New York, NY, 1996.

[48] J. Voas, A. Ghosh, G. McGraw, and K. Miller. "Gluing together software components: How good is your glue?", In Proc. of Pacific Northwest Software Quality Conference, Portland, October 1996.

[49] R. C. Cheung. "A user-oriented software reliability model". IEEE Trans. on Software Engineering, SE- 6(2):118–125, March 1980.

[50] J. C. Laprie , and K. Kanoun, "Handbook of Software Reliability Engineering", M. R. Lyu, Editor, chapter "Software Reliability and System Reliability", pp. 27–69. McGraw-Hill, New York, NY, 1996.

[51] B. Littlewood, "A semi-Markov model for software reliability with failure costs", In Proc. Symp. Comput. Software Engineering, pp. 281–300, Polytechnic Institute of New York, April 1976.

[52] V. S. Sharma, P. Jalote, and K. S. Trivedi, "Evaluating performance attributes of layered software architecture", In Proc. of 8th International SIGSOFT Symp. on Component-based Software Engineering (CBSE) St. Louis, Missouri, USA, May 2005.

[53] S. Krishnamurthy, and A. P. Mathur, "On the estimation of reliability of a software system using reliabilities of its components", In Proc. of Eighth Intl. Symosium on Software Reliability Engineering, pp. 146– 155, Albuquerque, New Mexico, November 1997.

[54] S. Yacoub, Bojan Cukic, and H. Ammar. "Scenariobased analysis of component-based software". In Proc. of Tenth Intl. Symp. on Software Reliability Engineering, Boca Raton, FL, November 1999.

[55] M. Xie, and C. Wohlin. "An additive reliability model for the analysis of modular software failure data". In Proc. Sixth Intl. Symp. on Software Reliability Engineering, pp. 188–193, Tolouse, France, October 1995.

[56] US Department of Defence Trusted Computer System Evaluation Criteria ("Orange Book"). http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html, December 1985. DoD 5200.28-STD.

[57] ISO/IEC International Standards (IS) 15408-1:1999, 15408-2:1999, and 15408-3:1999, Common Criteria for Information Technology Security Evaluation: Part 1: "Introduction and General Model", Part 2: "Security Functional Requirements", and Part 3: "Security Assurance Requirements", August 1999.Version 2.1 (CCIMB-99-031, CCIMB-99-032, and CCIMB-99-033).

[58] C. Landwehr, "Formal Models for Computer Security", Computer Surveys, vol. 13, no. 3, pp. 247–278, September 1981.

[59] J. Lowry, "An Initial Foray into Understanding Adversary Planning and Courses of Action", In Proc. of the DARPA Information Survivability Conference and Exposition II (DISCEX'01), pp. 123–133, 2001.

[60] K. Sallhammar, B. E. Helvik, and S. Knapskog, "A Framework for Predicting Security and Dependability Measures in Real-time", IJSNS International Journal of Computer Science and Network Security, vol. 7, no. 3, pp. 169-183,March 2007

[61] B. Littlewood, S. Brocklehurst , N. Fenton , P. Mellor, S. Page , D. Wright , J. Doboson , J. McDermid, and D. Gollmann, "Towards Operational Measures of Computer Security", Journal of Computer Security, Vol. 2, pp. 211-229, 1993.

[62] E. Jonsson, and T. Olovsson. "A Quantitative Model of the Security Intrusion Process Based on Attacker Behaviour", IEEE Transactions on Software Engineering, vol. 23, no. 4, pp. 235–245, April 1997.

[63] R. Ortalo, Y. Deswarte, and M. Kaˆaniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security", IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 633–650, 1999.

[64] Z. Zhang,F. Nait-Abdesselam, P. Ho, "Boosting Markov Reward Models for Probabilistic Security Evaluation by Characterizing Behaviours of Attacker and Defender", The Third Int'l Conf. on Availability,Reliability and Security, pp.352-359, 2008.

[65] B.B. Madan, K.S. Trivedi,"Security modeling and quantification of intrusion tolerant systems using attack-response graph." Journal of High Speed Networks, vol. 13, no. 4, pp. 297 – 308, 2004.

[66] K. Lye and J.M. Wing, "Game strategies in network security", Int'l Journal of Information Security, vol. 4, no. 1-2, pp. 71–86, 2005.

[67] K. Sallhammar, B. E. Helvik, and S.J. Knapskog, "Towards a stochastic model for integrated security and dependability evaluation", In Proc. of the First Int'l Conf. on Availability,Reliability and Security (ARES), pp. 156-165, 2006.

[68] P. Liu , and W. Zang, "Incentive-based modeling and inference of attacker intent, objectives, and strategies", ACM Transactions on Information and System Security (TISSEC), vol.8, no. 1, pp. 78 - 118,2005.

[69] W. Jiang, Z. Tian, H.L. Zhang, and X. Song, "A Stochastic Game Theoretic Approach to Attack Prediction and Optimal Active Defence Strategy Decision", pp. 648-653, April 2008.

[70] Y. Wang, C. Lin, and K. Meng, "Analysis of Attack Actions for E-Commerce Based on Stochastic Game Nets Model", Journal of computers, vol. 4, no. 6, June 2009.

[71] V. S. Sharma, and K. S. Trivedi," Architecture-Based Analysis of Performance, Reliability and Security of Software Systems", Proc. of the 5th international workshop on Software and performance, Spain, pp. 217 - 227 ,2005.

[72] F. Stevens, T. Courtney, S. Singh, A. Agbaria, J.F. Meyer, W.H. Sanders, and P. Pal, "Model-Based Validation of an Intrusion Tolerant Information System", Proc. 23rd Symp. Reliable Distributed Systems (SRDS 2004), pp. 184-194, Oct. 2004.

[73] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. "The M¨obius Framework and Its Implementation", IEEE Trans. on Software Engineering, vol. 28, no. 10, pp. 956–969, Oct. 2002.

[74] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, and P. Webster. "The M¨obius Modeling Tool", In Proc. of the 9th Intl Workshop on Petri Nets and Performance Models, pp. 241–250, September 2001.

[75] W. H. Sanders, and the Board of Trustees of the University of Illinois, "Mobius Manual", Version 2.2.1.

[76] E. Jonsson, "Towards an integrated conceptual model of security and dependability", Proc. of the First Int'l Conf. on Availability, Reliability and Security, pp. 646-653, 2006.

[77] M. W. Maier, "System and Software Architecture Reconciliation", Systems Engineering, vol. 9, no. 2, 2006.

[78] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "Pattern-Oriented Software Architecture, A System Of Patterns", 2000.

[79] E. Jonsson, T. Olovsson, "On the Integration of Security and Dependability in Computer Systems", IASTED Int'l Conf. on Reliability, Quality Control and Risk Assessment in Washington, Nov. 4-6, 1992.

[80] Perform Research Group, "UltraSAN User's Manual", 3.0 ed., Coordinated Science Laboratory, 1994.

[81] E. Jonsson and T. Olovsson. "A Quantitative Model of the Security Intrusion Process Based on Attacker Behaviour", IEEE Transactions on Software Engineering, vol. 23, no. 4, pp. 235–245, April 1997.

[82] M. Castro, and B. Liskov, "Practical Byzantine Fault Tolerance", In Proc. Of the Third Symp. on Operating Systems Design and Implementation, pp. 173–186, Debruary 1999.

[83] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pp. 382–401, July 1982.

[84] S. Hafezian, O.Das, "Security modeling of a layered system", Symp. on Information Assurance, Biometric Security and Business Continuity: Information Assurance and Privacy, pp. 296-301, Sep 2009.