PROBABILISTIC VALIDATION OF AN INTRUSION-TOLERANT REPLICATION SYSTEM

BY

SANKALP SINGH

B.Tech., Indian Institute of Technology, Kanpur, 2001

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

To my parents.

Acknowledgments

I would like to express my sincere gratitude towards my advisor, Prof. William H. Sanders, for his technical guidance throughout this research, and for the support and encouragement I have received from him. I would also like to thank Prof. Michel Cukier, at the University of Maryland, for insightful suggestions during the course of this research.

My officemates Vishu Gupta, James Lyons, Hari Ramasamy, and Fabrice Stevens have been an excellent support, providing me with both research insights and enjoyable discussions. I am thankful to all past and present members of the group, Adnan Agbaria, Jenny Applequist, Tod Courtney, Dave Daly, Salem Derisavi, Kaustubh Joshi, Sudha Krishnamurthy, Vinh Lam, Ryan Lefever, and Mouna Seri, for making the PERFORM group such a fun research group to be a part of. Particularly, I would like to thank Tod for all the help I received from him in solving my Möbius-related problems, and Jenny, for the help she has always been, especially in editing this thesis. I would also like to thank Franklin Webber and Partha Pal at BBN Technologies for feedback and discussions on early drafts of this work.

I am thankful to my roommates, Anand Shukla and Srikanth Kandula, for all their help and cooperation during our stay together at UIUC for the last two years.

Any words are too little to express my appreciation for the support my parents and brother have always given me throughout my studies. I am thankful to them for being a perpetual source of motivation and encouragement.

The research described in this thesis was funded by DARPA grant F30602-00-C-0172. I am grateful to Dr. Jaynarayan Lala and DARPA for providing direction to the ITUA project.

Table of Contents

List of Figures

List of Abbreviations

GCS Group Communication System

ITUA Intrusion Tolerance by Unpredictable Adaptation

ID Intrusion Detection

IDS Intrusion Detection System

SAN Stochastic Activity Network

UIUC University of Illinois at Urbana-Champaign

LAN Local Area Network

WAN Wide Area Network

Chapter 1

Introduction

The popularity of the Internet, electronic commerce, corporate networks, and distributed computing has caused a proliferation of critical distributed applications, the consequence of which is a high premium on survivability of these systems. The availability of valuable information on modern computer networks and our increasing dependence on various distributed applications have led to a proportionate increase in the complexity and variety of intrusions.

We define an *intrusion* into a computer system as an interaction with its interfaces that causes, and is *intended* to cause, the system to behave in a way its designers did not expect. Examples include 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. Some damage-causing activities that are not be included under this definition of "intrusion" include turning off the electricity, which is not an intrusion because it is not an interaction with the system's software interfaces; planting a trapdoor in the system's software during its development, which is not an intrusion (although it may enable future intrusions) because it is not an interaction with the system's software interfaces; and discovering a bug in the system during ordinary use, which is not an intrusion because there was no intention to cause unexpected behavior.

In this thesis, we will be concerned only with *malicious* intrusions. An intrusion is *malicious* if its intent is to cause a system to offer less-than-expected service or no service at all. Hence, this definition concentrates on denial-of-service intrusions only, and intrusions that merely probe a system's vulnerabilities are not considered malicious in this sense. Likewise, an intrusion whose only effect is to steal information from the system for the intruder is not considered malicious. (However, in both of those cases, an intrusion *would* be considered malicious if probing for vulnerabilities or stealing information eventually allows the intruder

to deny service.) Finally, an accident that leads to loss of service would not be a malicious intrusion, because it's not intentional and therefore is not an intrusion at all under our definition. From now on, we may use the term *attack* to refer to a malicious intrusion.

The traditional approach to handling intrusions is to prevent them, and a system that prevents intrusions is called *secure* [And72]. In the traditional approach, one designs a secure system by clearly specifying which behaviors are expected, and then carefully implementing the system to allow only the expected behavior.

Several factors make the traditional approach of developing secure systems less than satisfactory. The process of developing and validating a secure system has turned out to be extremely expensive, especially with the emergence of unbounded systems [BDF02] such as the Internet and large networked infrastructures. A secure system is hard to modify, because any modification requires an expensive re-validation. If a "secure" system turns out to be insecure, i.e., a flaw is found, there is no second line of defense against attacks.

Intrusion tolerance [BD90, DFF+88, DBF91, FDR94, FP85, EFL+97] is an emerging approach to security for such systems that aims to increase the likelihood that an application will be able to operate correctly in spite of malicious intrusions. The system does not prevent all intrusions, but it is designed to tolerate the effects of intrusions it cannot prevent.

In the intrusion tolerance approach, part of the system is considered *critical*, i.e., the part that is to be protected against the effects of successful intrusions. Typically, the critical part of the system will be one or more high-value applications whose services must not be interrupted. Other, non-critical, parts of the system may be expendable or protected less well.

An intrusion-tolerant system is not necessarily immune to every possible attack. Some attacks, of course, cause more damage than others, and some may cause too much damage to be tolerated. Therefore, intrusion tolerance does not guarantee that the critical part of the system will *always* function under attack, only that it is likely to function longer than a functionally equivalent system that is not intrusion-tolerant.

Because intrusion tolerance does not guarantee immunity to every possible intrusion, it must be a *quantitative* (i.e., "how much longer will the system function?") and *probabilistic* (e.g., "how likely is it that the system will perform for 2 more hours?") property of the system.

For those reasons, intrusion tolerance differs from (many) traditional security properties, which are (often) single predicates (i.e., the system either is secure or isn't secure), instead of being quantitative and probabilistic.

Before intrusion tolerance can be accepted as an approach to providing security, it is important to develop techniques to evaluate its efficacy. However, it is quite difficult to

reason about the correctness of security mechanisms. An alternative approach, which has received much less attention from the security community, is that of trying to probabilistically quantify the behavior of an attacker and his impact on the ability of the system to provide certain security-related properties. Due to the characteristics of intrusion tolerance as described above, this approach is particularly suitable for validating intrusion-tolerant systems. Probabilistic evaluation has been used extensively in the dependability community, but very few attempts have been made to use it to assess system security.

In this thesis, we show that probabilistic modeling using Stochastic Activity Networks (SANs) [MMS85, SM01] can be used for validating intrusion-tolerant systems. We demonstrate our approach by using SANs, implemented using Möbius [CCD+01, DCC+02], to model and validate an intrusion-tolerant replication system. The system modeled is a part of the Intrusion Tolerance by Unpredictable Adaptation (ITUA) architecture, which aims to provide a middleware-based intrusion tolerance solution. We attempted to build a model in a modular way, so that it could be easily adapted to a wide variety of intrusion-tolerant systems. We defined several measures on the model to characterize the intrusion tolerance provided by the system. We provide insights into the relative merits of various design choices by studying the variations in those measures in response to changes in system parameters.

The remainder of this thesis is organized as follows. First, Chapter 2 provides an overview of the related work in this area and outlines the motivation for our approach. Chapter 3 provides a brief overview of the ITUA replication system and the assumptions that were made in constructing the model. Chapter 4 describes the composed stochastic activity network representation of the model for the system described in Chapter 3. Chapter 5 gives the various results we obtained from the model, along with our interpretations and inferences. We conclude in Chapter 6 with a synopsis of the major contributions of the thesis. Appendix A provides the detailed code for the SAN models as implemented in Möbius.

Chapter 2

Related Work and Motivation

2.1 Traditional Approaches

Most traditional approaches to security validation have not been quantitative (e.g., the Security Evaluation Criteria [TCS85, ISO99]). Quantitative methods, when attempted, have either been based on formal methods [Lan81], and aimed to prove that certain security properties hold given a specified set of assumptions, or been quite informal, and used teams of experts (often called "red teams," e.g., [Low01]) to try to compromise a system. Both approaches, while being valuable in identifying system vulnerabilities, have their limitations.

2.2 Early Work on Probabilistic Validation

While most attempts to validate security properties have been non-probabilistic, a few attempts have been made to use probabilistic methods to assess system security. These efforts provide a useful starting point for our work.

Early work on probabilistic quantification of security was done by Littlewood et al. [LBF+93]. That work was exploratory in nature; in particular, their goal was to investigate the similarities between dependability and security in order to define measures of "operational security" similar to those used in dependability evaluation. In doing so, the authors made the important observation that effort, which may or may not be directly expressed in terms of time, was an appropriate unit for expressing security measures. The measures they defined included both the rate of occurrence of security breaches (similar to the rate of occurrence of failures) and the probability of service without a security breach given a specified level of effort by an attacker (similar to the definition of reliability). This work was very forward-looking; instead of providing concrete solutions, it focused primarily on suggesting questions that must be answered in order to make probabilistic security evaluation viable.

2.3 Quantitative Model from Attacker Behavior

Jonsson and Olovsson [JO97] presented a quantitative model of a security intrusion based on attacker behavior. When faced with a lack of data from which to build a model of a typical attacker, the authors conducted several experiments in order to gather such data. In building their resulting model, they postulated that a process representing the behavior of an attacker can be split into three phases: the learning phase, the standard attack phase, and the innovative attack phase. While that approach is a good step towards quantifying security probabilistically, it considers only one source of uncertainty in security validation: the behavior of the attacker.

2.4 High-level Models based on System and Attacker Behavior

Several attempts have been made to build models that take into account the attacker as well as the system being validated. We are aware of three such attempts. Gong et al. [GGPW+01] present a general 9-state model of an intrusion-tolerant system for describing known and unknown attacks; they do not explicitly represent vulnerabilities that can lead to the intrusions. The attacker and system are not represented explicitly in the model; instead, the model represents the state of the system in terms of (high-level) events that lead to failures (e.g., the sequence "good state," "vulnerable state," "triage state," "failure state" would represent a direct path to failure when an attack occurs). As such, the model is very high-level, and relies on a modeler to make the translation from the functional specification of a system to transitions in the 9-state model. Building on the above work, Madan et al. [MGPVT02] have used a semi-Markov model for quantitative evaluation of the security properties of an intrusion-tolerant system.

2.5 Scenario Graphs

Jha and Wing [JW01] propose that state-level modeling, formal logic, and a Bayesian analysis be used together to quantify the survivability of a system. The authors first model the network nodes and links of a networked system using state machines. Faults are then injected in the models; links are assumed to be non-faulty, and the nodes are assumed to be non-faulty, faulty, or intruded (compromised by an intruder). The third step consists of specifying a survivability property (e.g., the system enters a faulty state) using temporal logic. From

the state machine model with the injected faults and survivability property, the authors then generate a scenario graph. The scenario graph is then used for evaluating the overall system reliability or the latency using Bayesian networks.

2.6 Privilege Graphs

Finally, Ortalo et al. [ODK99] propose modeling known vulnerabilities in a system using a "privilege graph" (which is similar to a scenario graph, described above). By combining a privilege graph with simple assumptions concerning an attacker's behavior, the authors then obtain an "attack state graph." Parameter values for attack state graphs are presumed to have been obtained experimentally; once obtained, an attack state graph can be analyzed using standard Markov techniques to obtain several probabilistic measures of security. To illustrate the use of their approach, the authors present an analysis using data obtained from measurements taken on a large computer installation over a 21-month period. While the results of the case study are interesting, the approach that the authors take can do nothing more than assess the security of a particular system with respect to known vulnerabilities, and requires the collection of a large amount of data to populate a constructed model. Such an approach is thus well-suited to discovering and assessing the impact of known vulnerabilities in an operating system, but is less useful for predicting the relative efficacy of alternative intrusion tolerance techniques. In order to make such predictions, we need a higher-level approach that focuses on the operation of the intrusion tolerance mechanisms themselves, rather than on known vulnerabilities.

2.7 Ingredients of the Desired Approach

All the approaches described above provide good starting points for the development of a probabilistic approach to security validation. In particular, they suggest that measures similar to those used in dependability evaluation can be defined; that it may be possible to model attackers; and that the systems can be represented as state-level models in a way that captures either known or unknown vulnerabilities. The existing work also suggests that measurement can be used to quantify input parameter values in models related to known vulnerabilities. However, it does not provide a clear road map for comparing alternative intrusion tolerance approaches quantitatively, or for estimating the intrusion tolerance of particular approaches, particularly during the design phase and with respect to unknown vulnerabilities. We believe that our approach can achieve those goals. We now outline the

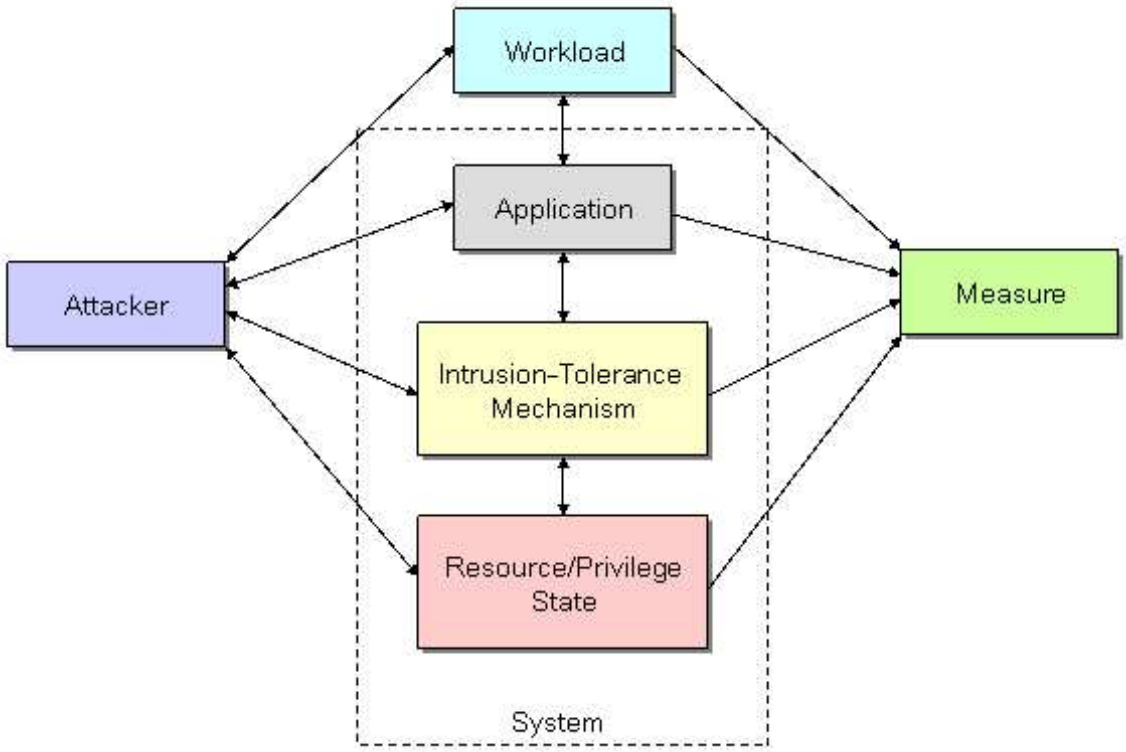


Figure 2.1: Probabilistic Security Model Structure

design principles of our approach, and highlight the major challenges we face.

Organizationally, we believe that a probabilistic validation of security with respect to availability should have two components: 1) a *model* of an attacker, the system, and the workload demanded of the system, that can be solved (either analytically or via simulation), and 2) a set of *measurements* that can provide estimates, accurate enough for the intended purpose of the model, of the values of model parameters.

More specifically, for intrusion-tolerant systems, our model would have the structure depicted in Figure 2.1. In particular, we believe that a probabilistic model of an intrusion-tolerant system can be broken down into five submodels: an *Attacker submodel*, a *Workload submodel*, an *Application submodel*, an *Intrusion-Tolerance Mechanism submodel*, and a *Resource/Privilege State submodel*. A model does not necessarily need to keep the submodels explicitly separate, but it should have that logical breakdown. The arcs connecting the submodels in the figure represent possible interactions between submodels that can change their state. For example, the attacker may be able to change the state of a resource or the amount of privilege granted to him (as represented by a directed arc from the attacker to the Resource/Privilege State submodel), or the attacker may change his state (and hence his behavior) by using knowledge he has gained by observing the state of the system (represented

by the directed arcs from the system submodels to the attacker.)

In each model, determining the appropriate level of detail/abstraction is very important, and depends on the scope and purpose of the model. Clearly, it is not possible to create a single universal model that is detailed enough to include all relevant abstractions, and still be solvable. Different models will be needed for different purposes and different attack classes. For a given model, the level of detail/abstraction that is appropriate will depend on many factors. For example, the system submodels should represent the parts of the system that are important, relative to the types of attacks considered and the expression of a particular availability measure. In particular, they must be detailed enough to support the expression of those parts of the state that an attacker may change, and those parts that may change his behavior. Depending on the nature of the attack, the attacker model may represent either the details of the intrusion itself (corresponding to explicit representation of faults in a dependability model) or the *effect* of the intrusion (corresponding to the representation of errors in a dependability model).

Likewise, system submodels must be detailed enough to support the expression of the availability measure considered. For example, Ortalo et al. [ODK99] suggest that an appropriate notion of state for a probabilistic system model would be the degree of privilege that an attacker has obtained. In addition to representing the degree of privilege, we believe that it is also important to represent resources in the system that are necessary for the application to function, since our ultimate goal is to quantify the availability as perceived by a user of an application. The two state aspects should be combined in the Resource/Privilege State submodel.

The level of detail/abstraction that should be employed when representing resources and privilege levels will depend on the type of intrusion tolerance that is employed to protect the system, and should be at least as fine-grained as a portion of the system in a single intrusion confinement boundary. The privilege level of a component within a system could be represented in many ways, ranging from a binary representation that is specific to the implementation of the component within the system (e.g., the attacker has obtained root access by exploiting a particular buffer overflow vulnerability on a particular OS version).

The level of detail/abstraction also depends on the input parameter values (obtained from measurement data) available for each model. The type and accuracy of input parameter values available will depend on the stage of development of the system that is being validated. Even if accurate input parameter values are not available, a model can still be used to study the trends in a system's security and availability for various parameter ranges, and the trends can be used to guide the system design process.

In this work, we show that probabilistic modeling using Stochastic Activity Networks

(SANs) [MMS85, SM01] addresses the above challenges. We demonstrate our approach by using SANs to model and validate an intrusion-tolerant replication system. The system modeled is a part of the Intrusion Tolerance by Unpredictable Adaptation (ITUA) architecture, which aims to provide a middleware-based intrusion tolerance solution. We attempted to build a model in a modular way, so that it could be easily adapted to a wide variety of intrusion-tolerant systems. We defined several measures on the model to characterize the intrusion tolerance provided by the system. We provide insights into the relative merits of various design choices by studying the variations in those measures in response to changes in system parameters.

Chapter 3

Overview of ITUA Replication System and Model Assumptions

The Intrusion Tolerance by Unpredictable Adaptation (ITUA) [CLR+02] architecture is a middleware-based intrusion tolerance approach that helps applications survive certain kinds of attacks. The ITUA architecture uses intrusion-tolerant group communication to eliminate single points of failure in processes and objects; integrates a set of COTS security tools that, together with the information from the group communication system, detect corrupt processes; and provides a decentralized replica management facility that decides what to do (in a possibly unpredictable way) when intrusions occur. ITUA assumes that as a result of an attack, replicas and management entities can fail in arbitrary ways. The management algorithms deal with the failure of management entities. We now describe the system as we have modeled it.

The system is divided into multiple security domains, each consisting of a set of hosts, as shown in Figure 3.1. Each domain implements a boundary that the attackers have difficulty crossing.

3.1 Managers

The decentralized management infrastructure of ITUA consists of architectural components known as *managers*. Each host runs a manager. There can be any number of applications, and the application objects protected by ITUA are replicated by the middleware and distributed across the security domains, subject to the constraint that a security domain can have only one replica from each application. We can think of various collections of objects as groups; the replicas of a replicated object form a replication group, and the managers of all security domains form a manager group. An intrusion-tolerant group communication

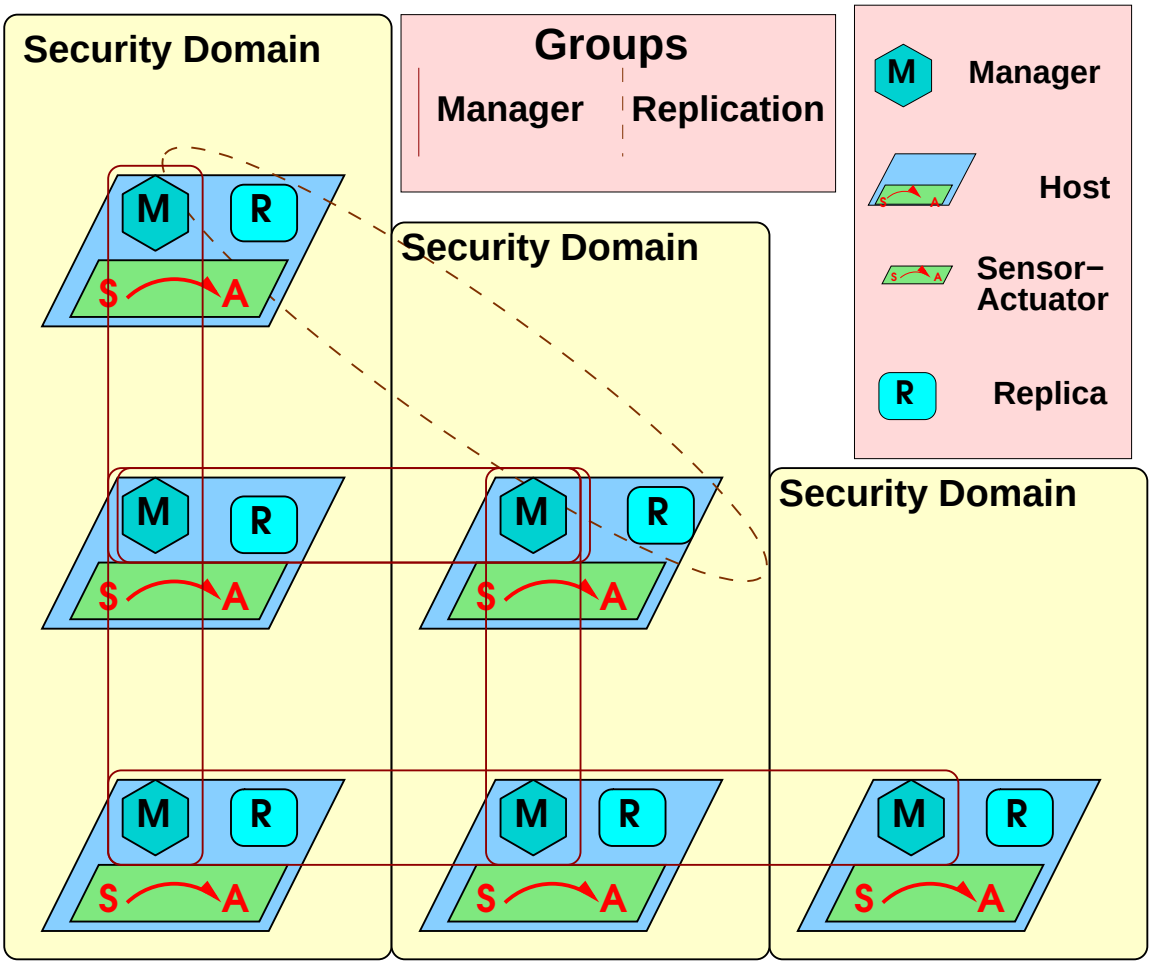


Figure 3.1: ITUA Architecure

system is used to multicast among replica groups and the manager group [RPL+02].

The ITUA replication management system has three major functions: making decisions, distributing information, and convicting entities. The first function of the management infrastructure is to make decisions about the group structure of the managers and replicas, such as deciding on the host on which to start new replicas. The management entities need to be aware of the state of the system in order to make decisions. The information may come from within a manager, from other managers sending what they know about the system, or from a replica making the manager on its host aware of a change. Hence, the second function is propagating information regarding important state changes among the managers. Finally, ITUA managers have the responsibility to convict corrupt members of the system. For example, when the system has decided that a member of the manager group is corrupt, the member is convicted, and the other members of the group will no longer communicate with it. The management entities enforce the conviction of the corrupt components, keeping

known corrupt processes from corrupting the system.

Frequently, the members of a group need to reach a consensus, either to convict a group member in a replica or manager group, or to help managers decide where to place a new replica. Since we consider all entities in the system susceptible to attacks that can potentially result in security breaches, there could be any number of entities in a group that are corrupt, but not yet detected. There is a limit on how many such undetected corruptions can be tolerated before the group becomes unable to reach consensus. We assume Byzantine fault tolerance [CL99, LSP82] using authenticated Byzantine agreement under a timed-asynchronous environment, and hence assume that less than a third of the currently active group members can be corrupt and still allow the group to reach consensus on various decisions. Note that group memberships are dynamic: some of the group members may have been killed upon detection of corruption, and new ones may have been started to replace them. Hence, the number of currently active group members may be less than the number of members the group initially started with, which would result in fewer Byzantine faults being tolerated.

3.2 Detection of and Recovery from Corrupt Replicas

We now describe how the management entities react when replicas become corrupt. The corruption of a replica can be discovered in two ways: by the intrusion detection software on its host, or, when the replica displays corrupt behavior during group communication, by other replicas in its replication group. The intrusion detection software can detect successful attacks against the host operating system and services, the replicas running on the host, or the manager running on the host. However, it cannot detect all such intrusions, and can even generate false alarms when there has been no actual security breach. On the other hand, we assume that when a corrupt replica behaves incorrectly during group communication (that uses Byzantine agreement), it is always detected and convicted by the correct members of the replication group through the use of authenticated Byzantine agreement, provided that the maximum number of undetected corruptions that the currently active group can tolerate has not yet been reached (i.e., less than a third of the currently active members are corrupt). If the other members of the replication group are able to reach a consensus on detection, they convict the corrupt replica, excluding it from all future communications. Each correct replica in this replication group then sends a message to the manager running on its host, informing it about the failure of the recently convicted replica. If a manager receiving such a message from a replica on its host is not itself corrupt, it multicasts the message to the

manager group. If there are enough managers to reach a consensus (i.e., less than a third of them are corrupt), they randomly pick the domain in which to start the new replica, with each currently active domain being equally likely to be picked. As mentioned earlier, the new domain cannot already have a replica of the application whose replica is being started. The managers within the chosen domain then randomly pick a host on which to start the replica (again with each host being equally likely to be picked) and the manager on the designated host then starts the replica.

When the intrusion detection software on a host detects an intrusion into either the host operating system or a replica running on the host, it informs the local manager. The further dissemination of this information and the subsequent exclusion of the host(s) and restarting of the replica(s) are similar to the response to detection by replica groups.

Under the current algorithm, the managers also convict the security domain that had the corrupt replica; they do so by excluding all the hosts in the domain, including their replicas and managers. That might result in restarting of some more replicas to replace the ones that were excluded. The motivation behind this preemptive approach is that when an entity on a host has been compromised, there is a good chance that other hosts in the domain have also been compromised, since the attacker may have been able to spread the attack to other hosts in the domain (perhaps by using techniques similar to those of the initial attack or using the corrupt hosts for covert purposes). Hence, the management system makes a preemptive strike by excluding the entire domain. When modeling the system, we have also considered an alternative approach in which only the host running the corrupt replica is excluded, not the entire domain. We have tried to study the ranges for system parameters (such as the ease of spreading an attack from one host in a domain to another) under which one approach is better than the other. We assume that the system is left to itself with minimum human intervention; hence, we do not model repair of excluded domains/hosts, and when the system has run out of domains (or hosts in the alternative approach) that can start a new replica of a particular application, no more replicas of that application are started.

3.3 Attacker Model

We make several assumptions about attacker behavior. We have based our attacker model on the experiments conducted by Jonsson and Olovsson [JO97], which suggest that there are three distinct classes of attacks: script-based attacks, more exploratory attacks, and totally innovative attacks. The script-based attacks are generally the most frequent, and are usually employed by inexperienced enthusiasts using scripts downloaded from the Internet.

The commercial intrusion detection software packages are regularly updated with information about the latest attack scripts and exploits; hence, we assume that the intrusion detection software can detect a fairly high percentage of script-based attacks. The next category are attacks from slightly more experienced attackers using intelligent combinations of various scripts. Those attacks are less frequent, but are also more difficult to detect. The third category is entirely new and innovative attacks. They are quite rare, but have an excellent chance of escaping detection. We also assume that attackers learn from successful intrusions. Thus, the corruption of a host in a security domain would increase the vulnerability of other hosts in the domain, as they probably have similar operating system versions and service configurations. We assume that an attacker can target the host operating system and services, application objects, or even the management infrastructure. We further assume that a successful intrusion into the host operating system greatly increases the chances of a successful intrusion into the application objects running on that host, as well as the manager on that host.

Since the focus of this paper is the intrusion-tolerant replication management architecture, rather than the underlying implementation, we assume the use of an intrusion-tolerant group communication system for multicasting within various groups. We also assume that a secure mechanism for starting replicas on a chosen host is in place. To keep the model simple, we assume that the middleware starts the same number of replicas for each application, and also that the security domains all have the same number of hosts.

Chapter 4

SAN Models

4.1 Stochastic Activity Networks

As stated in the introduction, we use Stochastic Activity Networks (SANs) ([MMS85, SM01]). SANs are a convenient, graphical, high-level language for describing system behavior. SANs are useful in capturing the stochastic (or random) behavior of a system. For example, we can almost always model fault arrivals by a random process.

We first describe stochastic Petri nets, which are a subset of SANs. A stochastic Petri net has the following components: *places* (denoted by circles), which contain tokens and are like variables; *tokens*, which indicate the "value" or "state" of a place; *transitions* (denoted by ovals), which change the number of tokens in places; *input arcs*, which connect places to transitions; and *output arcs*, which connect transitions to places. A transition is enabled if for each place connected by input arcs, the number of tokens in the place is greater than or equal to the number of input arcs connecting the place and the transition. When a transition is enabled, it may fire, removing a token from the corresponding place for each input arc and adding a token to the corresponding place for each output arc. An exponentially distributed time is assigned to each transition. The term *marking* of a place is used to indicate the number of tokens in the place.

Stochastic Petri nets, while being easier to read, write, modify, and debug than Markov chains, are still limited in their expressive power, since they may perform only +, -, >, and test-for-zero operations. That makes it very difficult to model complex interactions, and more general and flexible formalisms are needed to represent real systems. Stochastic activity networks are one such extension. They have many properties, which include a general way to specify that an activity (transition) is enabled, a general way to specify a completion (firing) rule, a way to represent zero-timed events, a way to represent probabilistic choices upon completion, state-dependent parameter values, and general delay distribution on activities.

SANs have all the components of Stochastic Petri nets plus four more: *input gates* (denoted by triangles pointing left), which are used to define complex enabling predicates and completion functions; *output gates* (denoted by triangles pointing to the right), which are used to define complex completion functions; *cases* (denoted by small circles on activities), which are used to specify probabilistic choices; and *instantaneous activities* (denoted by vertical lines), which are used to specify zero-timed events.

An input gate has two components: an enabling predicate and an input function. An activity is enabled if for every connected input gate, the enabling predicate is true, and for each input arc, the number of tokens in the connected place is greater than or equal to the number of arcs. Each case has a probability associated with it and represents a probabilistic choice of the action to take when an activity completes. When an activity completes, an output gate allows for a more general change in the state of the system than an output arc does. The output gate function is usually expressed using pseudo-C code. The input functions of all input gates connected to an activity are also executed when the activity completes. Those functions are also expressed in pseudo-C code. The times between enabling and firing of activities can be distributed according to a variety of probability distributions, and the parameters of the distribution can be a function of the state.

Composed models consist of SANs that have been replicated and joined multiple times. Replicated and joined models can interact with each other through a set of places (called *common* places) that are common to multiple submodels.

4.2 Composed Model

We have modeled the ITUA replication management system as a composed stochastic activity network. We built atomic SAN submodels for a single host, a single replica, and a management algorithm. We then built a complete model of the system using replicate and join operations.

Figure 4.1 shows the composed model for ITUA. The model consists of three atomic SAN submodels: **Replica**, **Host**, and **Management**. The **Replica** submodel is replicated `num_reps` times to form an *application* (a replication group) with `num_reps` replicas. The resultant submodel for an application is joined (*Join2* in the figure) with a **Management** submodel, which models the management algorithm for starting new replicas of the application when some are killed. The resultant submodel is replicated `num_apps` times (*Rep1* in the figure) to form `num_apps` applications (replication groups). Thus we model `num_apps` applications, each having `num_reps` replicated objects. Similarly, the **Host** submodel is replicated

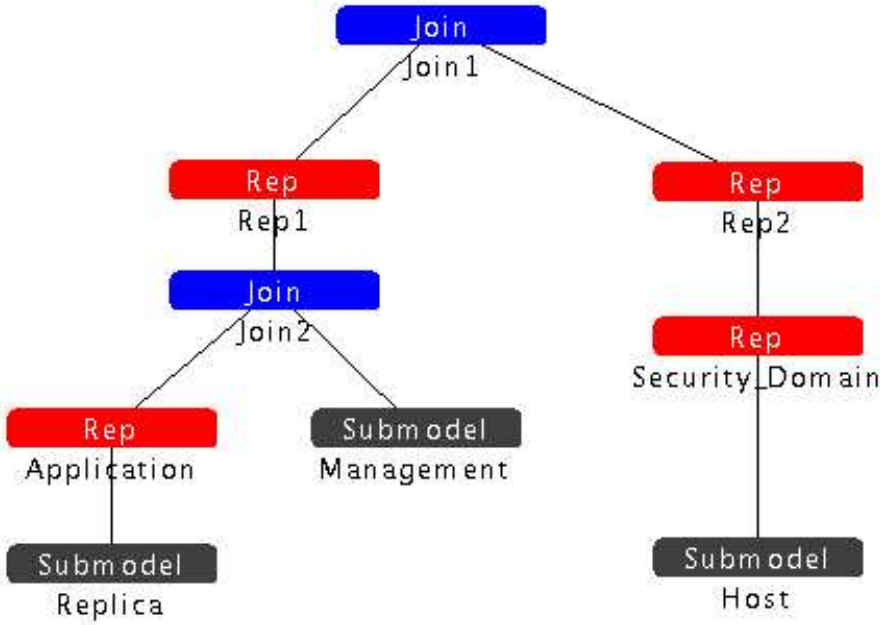


Figure 4.1: Composed Model

`num_hosts` times to form a *security domain*. The resultant security domain submodel is then replicated `num_domains` times (*Rep2* in the figure). Thus, we model `num_domains` security domains, each having `num_hosts` hosts. The models for applications and security domains are joined (*Join1* in the figure) to form the complete model for ITUA. The global variables `num_domains`, `num_hosts`, `num_apps`, and `num_reps` can be configured to any `short` integer value.

4.3 SAN Model for an Application Replica

Figure 4.2 shows the SAN representation of the **Replica** submodel. This SAN models the behavior of a single replica, including assignment of an application identifier, start of application replicas, attacks on the replica, detection and false alarms of the replica's corruption by the intrusion detection software, display of anomalous behavior by corrupt replicas, and the shutting down of the replica when the host on which it is running is shut down.

In order to create a correlation between replicas (represented by the *Replica* submodels) and the hosts (represented by the *Host* submodels) on which they are running, and to distinguish among replicas of different applications, we assign a unique identifier (which is a particular bit position in a bit vector) to each application. This is accomplished through

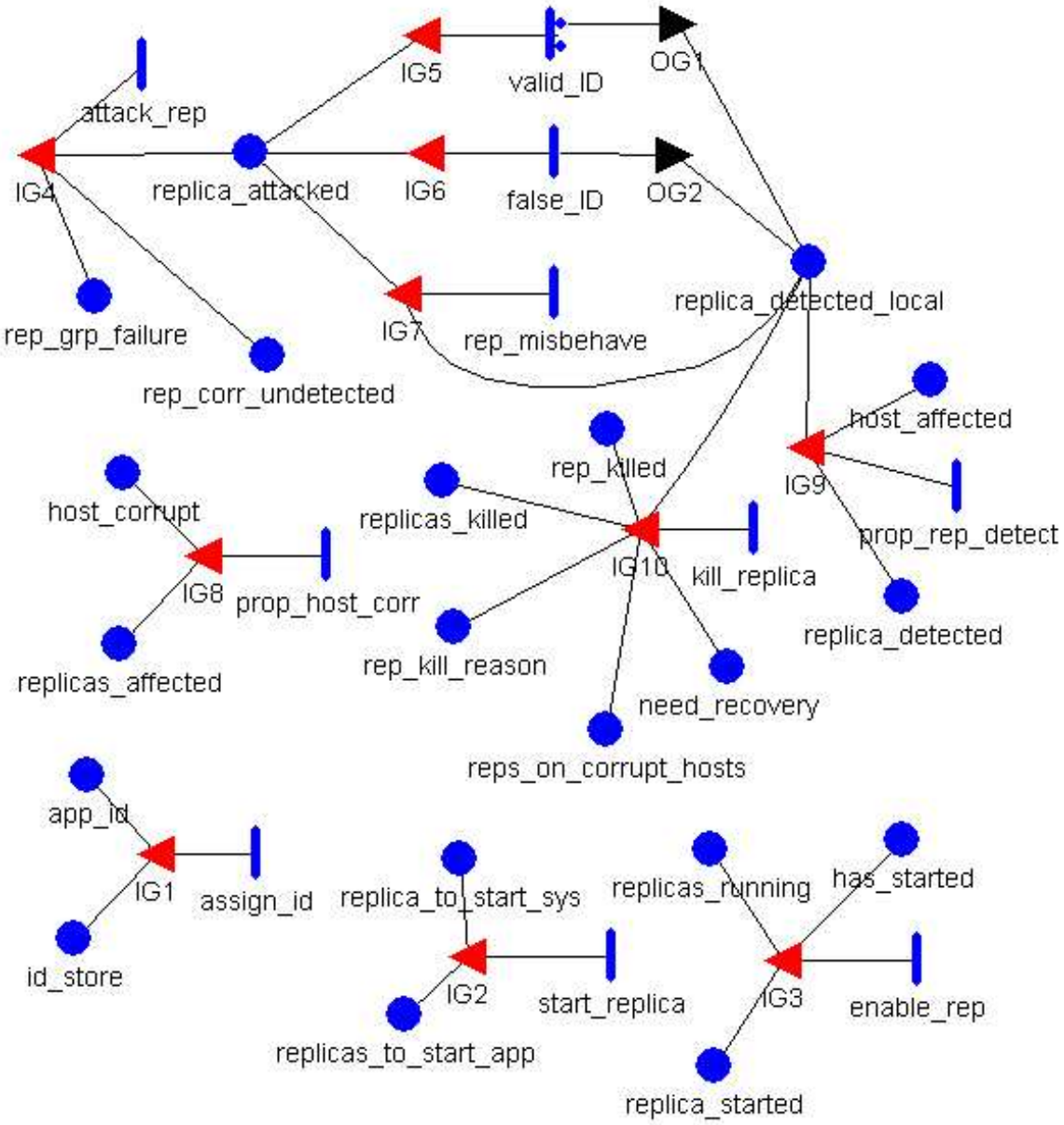


Figure 4.2: SAN Submodel for a Replica

the high-rate *assign_id* activity. The place *id_store* is assigned an initial marking of 2^{14}. The place *app_id* is shared by all replicas of an application, and stores the unique identifier for the application. The input gate *IG1* enables the activity *assign_id* if *app_id* has a marking of zero, i.e., no identifier has been assigned to the application. Upon the firing of *assign_id*, the input function in *IG1* assigns the current marking of *id_store* to *app_id* and changes the marking of *id_store* to half of the current value. The activity *assign_id* is configured with a very high rate, so that as soon as the model is solved or simulated, the applications quickly receive unique identifiers of the form 2^n, $0 \leq n \leq 14$. That assignment of identifiers limits the value of `num_apps`, the number of applications, to at most 15, but we believe that is more than enough for most studies.

The marking of place *replicas_to_start_app* is initialized to the global variable `num_reps`, which is the number of times each Replica submodel is replicated to form an application. *replicas_to_start_app* is shared across all *Replica* submodels that form an application. The activity *start_replica* can fire as long as the marking of *replicas_to_start_app* is not zero (i.e., there are replicas to be started). Upon firing, it puts a bit 1 at the place indicated by the application's identifier (stored in *app_id*) in the place *replica_to_start_sys*, which is shared by all the SANs in the composed model. Hence *replica_to_start_sys* is a 15-bit vector with 1s for all applications whose replicas are to be started. It is used by the *Host* submodels to start the corresponding replicas.

When a host starts some replicas, it puts a bit vector with 1s for all the applications whose replicas were started in the globally shared place *replica_started*. For each such application, the *enable_rep* activity is enabled in all *Replica* submodels that correspond to the application's replicas that have not started yet, with each enabled activity being equally likely to fire first. The first *enable_rep* activity to fire 1) increments the marking of *replicas_running*, which is shared across all replicas of an application and keeps track of the number of currently running replicas of the application; 2) sets the marking of *has_started*, which is local to its *Replica* submodel and indicates if the replica represented by the submodel is active, to 1; and 3) removes the application's bit from *replica_started*. Thus ,when a replica of an application is started on a host, one of the *Replica* submodels that belong to the application is randomly chosen, with each eligible model equally likely to be chosen to be the replica started.

Whenever a host becomes corrupt, the *Host* submodel puts a bit vector of application identifiers of all the replicas running on it in the globally common place *replicas_affected*. That is possible because a host (and for some management schemes, the entire domain) can have at most one replica of a particular application. The place *host_corrupt* is local to the submodel and indicates if there has been an intrusion into the host on which the replica represented by this *Replica* submodel is running. The activity *prop_host_corr* is enabled

whenever the bit vector in *replicas_affected* contains a 1 for the application to which this *Replica* submodel belongs and the replica is not already running on a corrupt host. When the activity fires, the bit for this application in *replicas_affected* is reset, and the marking of *host_corrupt* is set to 1.

The activity *attack_rep* represents a successful attack on a replica. As for all other activities in the model, the time between enabling and firing of this activity is exponentially distributed. The rate of this activity (i.e., the reciprocal of the mean time between firings of the activity when it is enabled) is higher if the replica is running on a corrupt host (indicated by the marking of *host_corrupt*). We multiply the base rate by a constant factor if the host is corrupt to obtain the higher rate. For most studies, the constant factor is set to 2, though we also experiment with other values. The factor of 2 implies that if there is an intrusion into the host on which a replica is running, the replica becomes twice as vulnerable to attacks as it originally was. Upon firing of *attack_rep*, the marking of the local place *replica_attacked* is set to 1 to indicate the intrusion, and the marking of *rep_corr_undetected*, which is shared for all replicas of the application, is incremented to indicate the number of yet-undetected corrupt replicas of the application. The replication group is checked for a Byzantine failure; if the number of undetected corrupt replicas is greater than a third of the total number of application replicas currently running (a Byzantine failure is found) the marking of *rep_grp_failure*, which is shared across all replicas of the application, is set to 1. *rep_grp_failure* is used to determine the "unreliability" of the application.

After an intrusion into a replica, the activities *valid_ID* and *rep_misbehave* are enabled. The activity *valid_ID* has two cases, which represent successful detection and failure to detect, respectively. Like the rates of most of the activities, the probability of successful detection is a configurable parameter of the model. Upon successful detection indicated by the firing of *valid_id*, the output function in gate *OG1* is executed; that decrements the marking of *rep_corr_undetected* and sets the marking of *replica_detected_local* to 1.

A corrupt replica may exhibit anomalous behavior, which can be detected by other currently running replicas of the application provided that there are enough correct members in the replication group for the application. Input gate *IG7* and activity *rep_misbehave* capture this. After a successful intrusion into the replica, the activity is enabled provided that the value of the marking of *replicas_running* is more than three times the value of the marking of *rep_corr_undetected* (i.e., less than a third of the currently running replicas are corrupt). The result of firing *rep_misbehave* is similar to the firing of *valid_ID*.

The false alarms of replica corruption raised by the intrusion detection system are represented by the activity *false_ID*. It is enabled whenever the replica has not been intruded. The result of firing *false_ID* is similar to that of firing *valid_ID*.

The activity *prop_rep_detect* conveys information about detection of the replica (through valid intrusion detection, false alarm, or detection of misbehavior by the replication group) to the host on which the replica is running. The places *rep_detected* and *host_affected* are shared across all SANs in the composed model. Upon activity firing, the marking of *rep_detected* is set to the application identifier of the application to which the replica belongs, and the marking of *host_affected* is set to the marking of *host_corrupt*, to help in the choice of a proper *Host* submodel as the model of the host running the replica. The marking of *replica_detected_local* is set to 2, to indicate that the information on replica corruption has been propagated to the rest of the system and to keep *prop_rep_detect* from firing repeatedly.

When a host (or domain) is shut down (excluded), all the replicas running on the host (or domain) are killed. The fact that they have been killed is conveyed from the *Host* submodel to the *Replica* submodels through the globally shared places *replicas_killed*, *rep_kill_reason*, and *reps_on_corrupt_hosts*. The marking of *replicas_killed* is a bit vector with a 1 for all the applications whose replicas were killed. The marking of *rep_kill_reason* is a bit vector with a 1 for all the applications that had compromised replicas on the host (or domain). The marking of *reps_on_corrupt_hosts* is also a bit vector with a 1 for all the applications that had their replicas running on corrupt hosts in the domain being shut down. The markings of those places are used to determine the appropriate *Replica* submodels to kill (in other words, the appropriate conditions to put in the predicate of gate *IG10*). Hence, the activity *kill_replica* is fired only in the appropriate *Replica* submodels. Upon firing of *kill_replica*, the appropriate bits in *replicas_killed*, *rep_kill_reason*, and *reps_on_corrupt_hosts* are reset, the marking of *replicas_running* is decremented, the markings of various local places in the *Replica* submodel are reset so that the submodel can be used again to start a new replica of the application, and the marking of *need_recovery* is incremented to indicate that the management infrastructure must start a new replica for this application.

4.4 SAN Model for Management Algorithm

Figure 4.3 shows the SAN representation of the **Management** submodel. This SAN models the process of recovery by the management infrastructure through the starting of new replicas to replace those killed due to domain and host exclusions. As shown in Figure 4.1, there is one *Management* SAN per application. The place *need_recovery* is shared by all the replicas of the application. Its marking indicates the number of replicas that need to be started for the application. The place *app_id* is also shared with the application, and, as described before, it contains the unique identifier for the application. The other

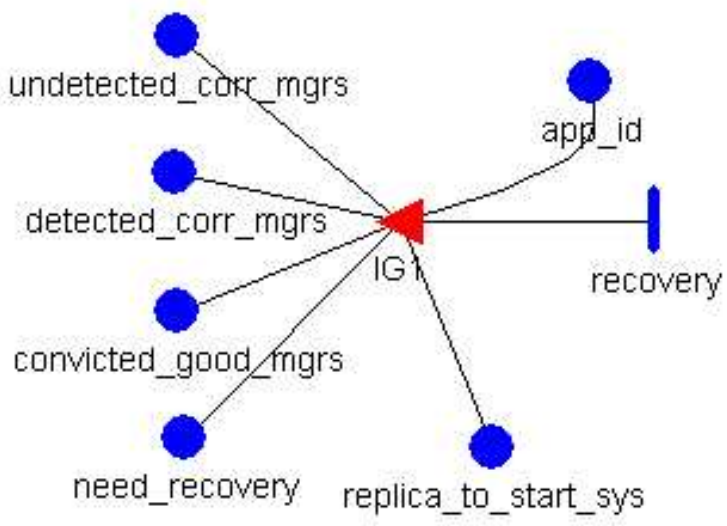


Figure 4.3: SAN Submodel for the Management Algorithm

three places are shared by all SANs in the composed model. The activity *recovery* is enabled whenever there are some tokens in *need_recovery* and there are enough good managers in the system to initiate a recovery. The latter condition holds when the total number of managers is greater than three times the number of undetected corrupt managers ($num_domains - (detected\_corr\_mgrs \rightarrow marking + convicted\_good\_mgrs \rightarrow marking) > 3 * undetected\_corr\_mgrs \rightarrow marking$). Upon the firing of *need_recovery*, the application's identifier is placed in the *replica_to_start_sys* place, which is then used by *Host* SANs to start the replica.

4.5 SAN Model for a Host

Figure 4.4 shows the SAN representation of the **Host** submodel. This SAN models the activities on a single host, such as attacks on the host, detections and false alarms by the intrusion detection software on the host, starting of replicas on the host, starting of management entities on the host, and shutting down of the host and all replicas it is running.

The activity *start_manager* is a high-rate activity that is responsible for starting a manager in each domain. The place *managers_to_start* is shared across all hosts in a domain, and its marking is initialized to 1. The place *managers_started* is shared across all hosts in all domains and is used to keep track of the number of domains that have managers up and running. The place *has_manager* is local to the *Host* SAN, and is set to 1 upon firing of *start_manager*, which also decrements the marking of *managers_to_start* to zero.

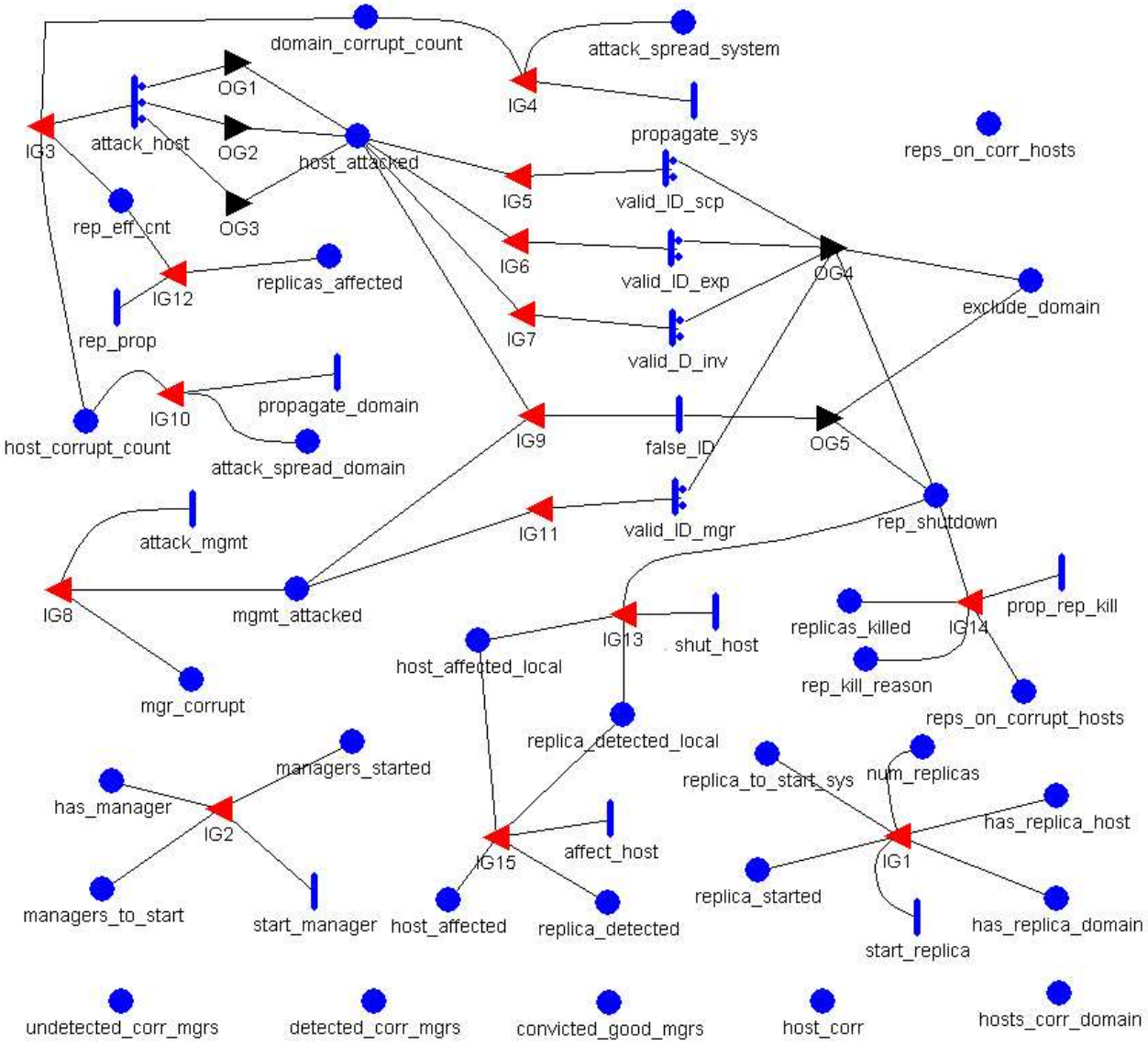


Figure 4.4: SAN Submodel for a Host

The high-rate activity *start_replica* represents the process of starting replicas on hosts. As mentioned during the description of the *Replica* SAN, *replica_to_start_sys* is shared across all SANs in the composed model, and its marking is a bit vector with 1 for all the applications whose replicas are to be started. Activity *start_replica* is enabled whenever there is at least one application for which there is a replica to start and for which there is not already a replica in the domain. Upon the firing of *start_replica*, replicas for all the applications that had replicas to start and did not have replicas in the domain are started on the host. The local place *has_replica_host*, which holds a bit vector of all application identifiers that have replicas running on the host, and the place *has_replica_domain*, which is shared across all hosts in the domain and holds a bit vector of all applications with replicas in the domain, are updated accordingly. The marking of *replica_to_start_sys* is updated to reset bits for all applications whose replicas were just started. The local place *num_replicas* is updated to represent the number of replicas (of all applications) running on this host. Information identifying the replicas for which applications were started is put as a bit vector into the place *replica_started* (which is common across all SANs), and is then used to enable appropriate *Replica* submodels. The globally common place *host_is_corrupt* is used to convey the corruption state of this host to the replicas that would be enabled.

As mentioned in Chapter 3, the attacker can attack the host (i.e., the host operating system and services), the management infrastructure, and the application replicas. We have already described (during the description of the *Replica* SAN) how attacks on replicas are modeled. The activity *attack_host* models attacks on the host operating system and services. As mentioned in Chapter 3, an attack on the host can belong to one of three categories: script-based, more exploratory, and innovative (which are listed in order of decreasing frequency and decreasing probability of detection). The three categories are modeled by three *cases* for the activity *attack_host*, which are represented by small circles on the activity. The distribution of the cases is a configurable parameter, and is set to provide decreasing probabilities to cases associated with script-based attacks, exploratory attacks, and innovative attacks. Upon the firing of *attack_host*, a function in the output gate that corresponds to the chosen case is executed; that sets the marking of *host_attacked* to be 1, 2, or 3 depending upon the case chosen.

Information about the intrusion into the host needs to be conveyed to replicas running on the host, since the intrusion affects their vulnerability (that is, the rate of *attack_replica* in the *Replica* SAN). The information is conveyed through the activity *rep_prop*. Upon firing of *attack_host*, the marking of the local place *rep_eff_cnt* is set to 2, which enables *rep_prop*. Upon firing of *rep_prop*, the marking of globally common place *replicas_affected* is set to a bit vector that represents all the applications that have replicas on this host. It is then used

by the *Replica* SANs as already described. The marking of *rep_eff_cnt* is set to 1 to prevent *rep_prop* from firing again.

As mentioned in Chapter 3, successful intrusion into a host increases the vulnerability of other hosts in the system, especially the ones in the same security domain. We model this by including two "propagate" activities: *propagate_domain* and *propagate_sys*. The activity *propagate_domain* models the spread of an attack within a domain, and *propagate_sys* models the spread of an attack across domain boundaries. Upon the firing of *attack_host*, the marking of *host_corrupt_count* is set to 1, enabling *propagate_domain*. Upon the firing of *propagate_domain*, the marking of place *attack_spread_domain*, which is shared by all hosts in the domain, is incremented by a model variable representing the amount of spread effect. This variable also determines the rate of the *propagate_domain* activity. The marking of *host_corrupt_count* is set to 2 to prevent the activity from firing again. The activity *propagate_sys* is handled similarly, except that *attack_spread_system* is shared across all hosts in all domains. The rate of the activity *attack_host* is proportional to the markings of *attack_spread_domain* and *attack_spread_system*. Hence, higher spread increases the chances of successful intrusions into the host operating system and services. The spread effect variable associated with in-domain spread is set to be much larger than the variable associated with across-domain spread.

The activity *attack_mgmt* represents attacks against the manager running on the host. Upon the firing of *attack_mgmt*, the marking of the local place *mgmt_attacked* is set to 1. The marking of *mgr_corrupt*, which is shared by all hosts in the domain, is set to 1 if a third or more of the active managers in the domain have been corrupted. The marking of *undetected_corr_mgrs*, which is shared by all SANs in the composed model, is also incremented.

The activities *valid_ID_scp*, *valid_ID_exp*, and *valid_ID_inv* represent the detection by the intrusion detection software of infiltration into the host OS and services for script-based, more exploratory, and innovative attacks, respectively. Each activity has two cases, which represent successful detection and failure to detect. In most studies, the probability of successful detection is usually set to be much higher for script-based attacks than for exploratory and innovative attacks. Similarly, the probability of successful detection is set to be much higher for exploratory attacks than for innovative attacks. The activity *valid_ID_mgr* represents successful detection by the intrusion detection software of infiltration of the management entity on the host. Upon firing of any of the detection activities, the function in gate *OG4* is executed, provided that the manager on the host and the manager group of the domain are not corrupt. It sets the marking of *exclude_domain* to 1. *exclude_domain* is shared for all hosts in the domain, and indicates that all the hosts in the domain have been excluded from the system. Suitable changes are made to the markings of the globally common places

undetected_corr_mgrs and *detected_corr_mgrs*. If no replica on the host is corrupt (as indicated by *replica_detected_local*) then the marking of *rep_shutdown* (shared for all hosts in the domain) is set to 1. If there are corrupt replicas, it is set to 2.

False alarms of infiltration into the host OS or host's management entity are represented by the activity *false_ID*, which is enabled as long as there have not been any actual intrusions. Upon the firing of *false_id*, an action similar to those taken upon the firings of various valid detection activities is taken. Changes are also made to the marking of *convicted_good_mgrs*, if necessary.

As mentioned during the description of the *Replica* SAN (Section 4.3), when either the intrusion detection software or replication group members find a replica to be corrupt, the identifier of the application is put in the globally shared place *replica_detected*. The place *host_affected* has a marking of 1 if the host on which the detected replica was running was corrupt. The activity *affect_host* is enabled if that host has a replica of the application indicated by *replica_detected* and its corruption status matches with that conveyed by *host_affected*. Upon the firing of *affected_host*, *replica_detected* and *host_affected* are reset, and their values are transferred to the places *replica_detected_local* and *host_affected_local*, which are shared by all hosts in the domain. This is done to avoid the possibility of a deadlock in the model. The activity *shut_host* is enabled if there is some corrupt replica (i.e., a non-trivial marking for *replica_detected_local*), and either the domain's manager group is not corrupt or there are enough good managers in the system. The reason for the latter condition is that even if the domain's manager group is corrupt and does not report the intrusion and exclude itself, other managers would know about the corrupt replica from the other members of the replica's replication group. If there are enough good managers (i.e., no more than a third are in the undetected corrupt state), then they will exclude the corrupt manager and its domain. The marking of *rep_shutdown* is set to 2, and the marking of *exclude_domain* is set to 1.

The actual shutting down of the domain is modeled through the activity *prop_rep_kill*. The activity *prop_rep_kill* is enabled when *rep_shutdown* has a non-zero marking. Upon the firing of *prop_rep_kill*, the marking of *replicas_killed* is set to be the bit vector stored in *has_replica_domain*, which indicates the applications that have replicas in this domain. As mentioned before, the marking of *rep_kill_reason* is set to be the bit vector that represents the applications that had corrupt replicas in this domain, and the marking of *reps_on_corrupt_hosts* is set to the bit vector representing the applications that had replicas running on corrupt hosts in the domain. Those two places are globally common, and are used by *Replica* SANs to shut down appropriate replicas.

We have also modeled an alternative management algorithm that excludes from the sys-

tem only the host on which infiltration is detected (and kills only the replicas running on that host), instead of excluding the entire domain. The SANs for that approach look almost the same, but have a few subtle differences from the SANs described above, with respect to the places that are shared, the levels at which they are shared, and the input predicates and functions that are used. Specifically, the places *exclude_domain*, *rep_shutdown*, *replica_affected_local*, and *host_affected_local* were made local to the *Host* SAN. Firing of *prop_rep_kill* now sends information about replicas on the host (if the host was corrupt). (In the previous model, this information was sent about the domain.) We made the corresponding changes to the input function in gate *IG10* in the *Replica* SAN.

Chapter 5

Results

We used the Möbius [CCD+01, DCC+02] tool to design the SANs, define the intrusion tolerance measures on the model, and design studies on the model. Möbius can solve SANs analytically by converting them into equivalent continuous time Markov chains. However, because of the complexity of the model and the use of non-exponentially distributed firing times for some activities, we instead used Möbius to simulate the model to obtain values for the intrusion tolerance measures for various studies.

We defined several measures on the model for use in the studies. They include:

- *Unavailability* for an interval, which denotes the fraction of time the service was improper in the interval. Service by an application is defined to be improper if it suffers a Byzantine fault, i.e., a third or more of the currently running replicas are corrupt.
- *Unreliability* for an interval, which denotes the probability that service was improper at least once in the interval, with improper service as defined above.
- *Number of replicas* of an application still running at a given time instant.
- *Number of replicas per host* or the load on a host at a given time instant.
- *Fraction of corrupt hosts* in a domain at the instant of time it is excluded.
- *Fraction of excluded domains* at a given instant of time.

We now describe the studies we conducted using the model. To determine the preferable distribution of hosts into domains, we compared the intrusion tolerance of the system for different distributions of a constant number of hosts into domains, as well as for different numbers of hosts distributed into a fixed number of domains. Another study compared the relative efficacy of host-exclusion and domain-exclusion management algorithms. We studied the effect of the rate of false alarms raised by the intrusion detection software on the quality

of intrusion tolerance provided by the system. A study evaluating the effect of how quickly infiltrated replicas depict corrupt behavior during group communication on the intrusion tolerance of the system was also performed.

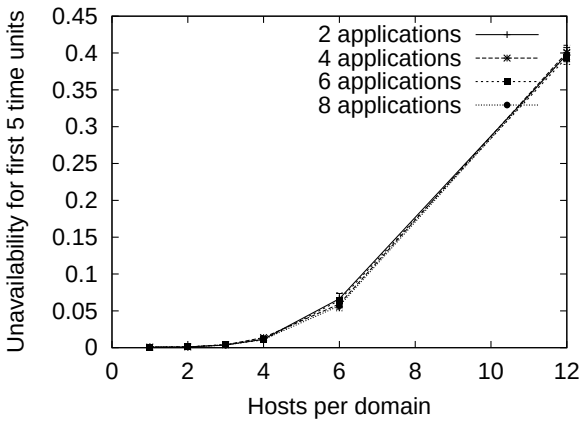
Unless otherwise specified, the following values were used for the parameters of interest in the studies described in this section (for ease of understanding, consider one time unit = one hour):

- Cumulative base attack rate on the system was 3 successful attacks per time unit. Since the actual attack rates can increase as a result of various factors, such as attack spread, corruption of the host on which a replica or a management entity is running, and other causes, the actual attack rate will usually be higher than this.
- Cumulative false alarm rate was 2 false alarms per time unit.
- Distribution of attack on a host (OS and services): 80% script-based, 15% more exploratory, and 5% innovative.
- Intrusion detection probabilities: 90% for script-based on hosts, 75% for more exploratory on hosts, 40% for innovative on hosts, 80% for replicas, and 80% for management entities.
- *Miscellaneous*: domain propagation rate of 1 per time unit, system-wide propagation rate of 0.1 per time unit; infiltration of a host doubles the chances that the replicas and management entity running on it will also be infiltrated; a corrupt replica exhibits anomalous behavior twice per time unit.

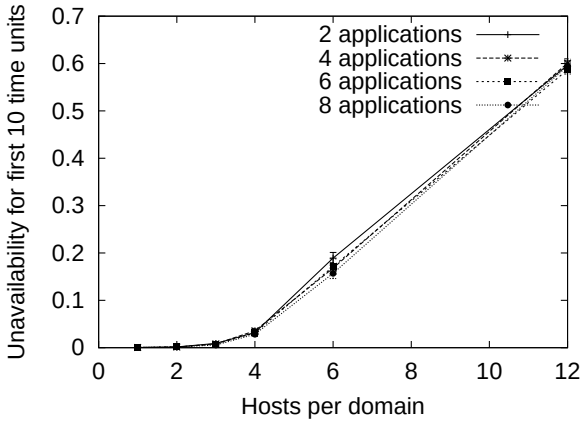
The above values are based on experience and the security literature. In the studies that follow, we have also tried to explore the system's sensitivity to variations in these parameters. All the results presented have a 95% confidence interval, and the bounds on the estimates of the measures have been shown using vertical bars on all the graphs. The bars may not be discernable due to the high accuracy of the simulation in some cases.

5.1 Comparative Intrusion Tolerance Under Different Distributions of a Constant Number of Hosts into Domains

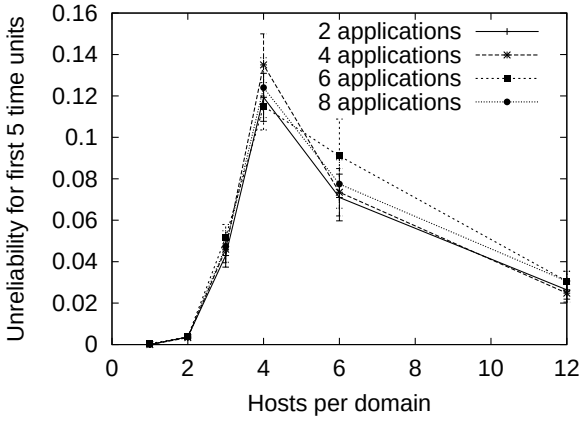
ITUA is a system in active development. A major design issue is that of deciding how many hosts to put in each security domain. Is it better to consider each host as a separate



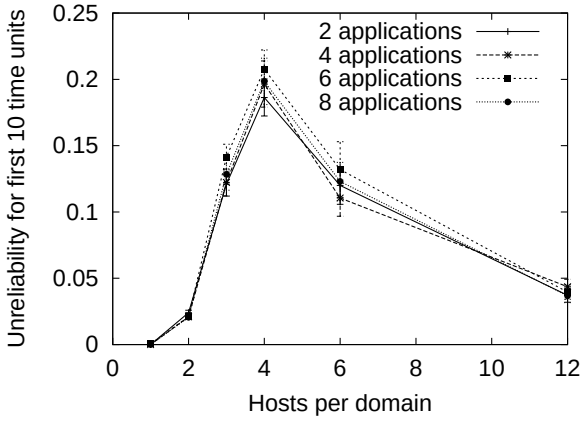
(a) Unavailability after 5 hours



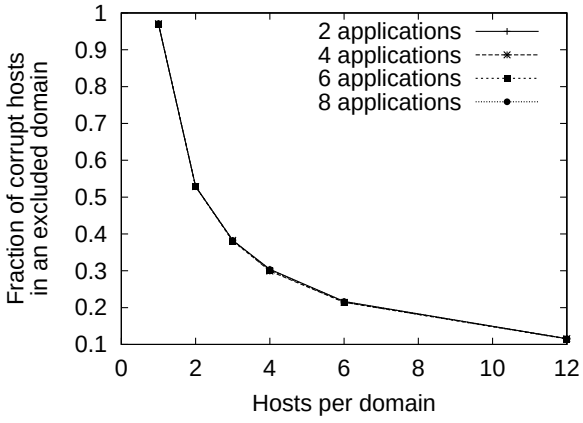
(b) Unavailability after 10 hours



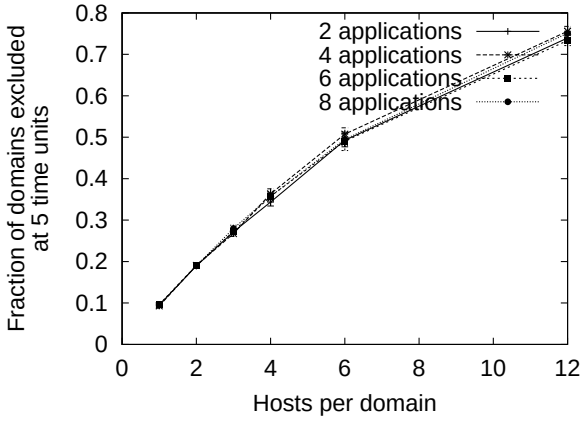
(c) Unreliability after 5 hours



(d) Unreliability after 10 hours



(e) Fraction of Corrupt Hosts in an Excluded Domain



(f) Fraction of Domains Excluded

Figure 5.1: Variations in Measures for Different Distributions of 12 Hosts for First 5 hours

security domain than to consider multiple hosts within a security domain? To answer that question, we designed two studies. The aim of the first study was to see how the system performed for various distributions of a constant number of hosts. We conducted experiments by distributing 12 hosts into 1, 2, 3, 4, 6, or 12 domains. For each distribution, we considered 2, 4, 6, or 8 applications with 7 replicas each.

Figures 5.1(a) and 5.1(b) shows the variation of unavailability of an application for different distributions of the 12 hosts. An important point to note is that as we move along the X-axis, the number of domains reduces as the number of hosts per domain increases. As is evident from the graphs for unavailability for the first 5 time units (hours), the system is more available when we have fewer hosts per domain, mostly because fewer hosts allow for more domains, so that we do not run out of domains when many of them have been excluded. Other points of interest are that the unavailability is low even when the system is left without any human intervention for a few hours. We also note that unavailability for a particular application does not change much with an increase in the number of applications.

Figures 5.1(c) and 5.1(d) shows the variation of unreliability of an application for different distributions of 12 hosts. The relevant portion of this graph is between $x = 0$ and $x = 4$. This portion clearly shows that unreliability increases rapidly as we increase the number of hosts in a domain and (since the total number of hosts is constant) decrease the total number of domains. The maximum unreliability occurs at 4 hosts per domain and then decreases for higher numbers of hosts. This is explained by the classic reliability argument, which states that if the failure of one replicated component can cause a catastrophic failure, then increasing the replication would decrease the reliability, since the chances of one failure would increase. With four or more hosts per domain, there are enough replicas to form 1, 2, or 3 domains. Consequently, for any application we would be able to run at most 1, 2, or 3 replicas respectively (since we can have only one replica per domain for any application). In each of the scenarios, corruption of one replica would result in a failure to reach Byzantine agreement, and the chances of such corruption would be higher when we have more replicas.

Figure 5.1(e) shows the fraction of hosts that were infiltrated in a domain by the time it was excluded. Resources are wasted when we have more hosts per domain, since in that case corruption of a very small fraction of hosts results in exclusion of a large number of uncorrupted hosts. Note that the fraction is not 1 when we have one host per domain, since false alarms can result in some domains being excluded without any host being corrupted. Figure 5.1(f) shows the fraction of domains that have been excluded at the end of 5 hours. Again, we must point out that as the number of hosts per domain increases, the number of domains decreases, as we have a constant number of hosts (12). Once again, we notice that a large number of domains were excluded when we had more hosts per domain, adversely

affecting the availability of the applications.

5.2 Comparative Intrusion Tolerance Under Different Numbers of Hosts Distributed into a Constant Number of Domains

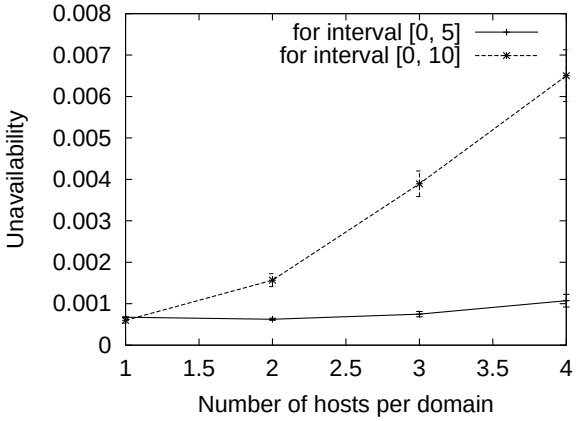
The above studies indicate that for a given set of hosts, it is best to distribute them into as many domains as possible. To determine the gains to be obtained by putting more hosts in each domain, and to judge the cost/benefit ratio, we conducted a second study in which the number of domains was fixed at 10 and the number of hosts per domain was varied from 1 to 4. The study had 4 applications, each with 7 replicas. Other parameters were similar to those in the previous study. Hence, in the second study, the total number of hosts changes for each experiment in the study.

Figures 5.2(a) and 5.2(b) show, respectively, the variation in unavailability and unreliability with an increasing number of hosts in 10 domains. Since the probability of a successful intrusion into a host is assumed to be the same in all experiments, the existence of more hosts in a domain implies a greater chance that one of them will be corrupt, resulting in exclusion of the entire domain. This causes a slight increase in the unavailability and unreliability. Note, however, that the variation in values for the first five time units is quite low, and even for the first ten time units, it is much lower than the variation observed in experiments in Section 5.1.

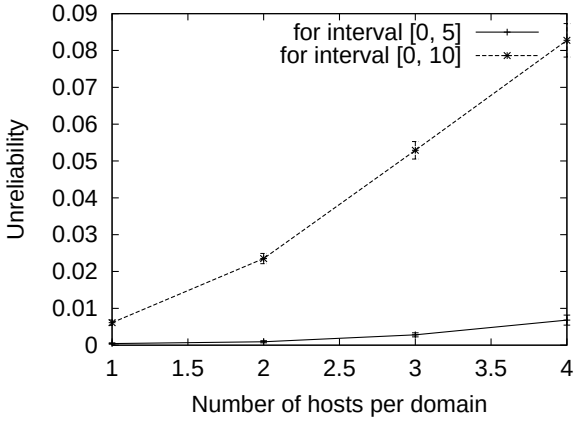
Figure 5.2(c) shows that a considerable waste of resources takes place when we put more hosts in a domain, since the domain will be excluded as soon as a small number of hosts are corrupted. Figure 5.2(d) indicates that the number of domains that have been excluded increases due to the increase in the number of hosts. This is again explained by the fact that corruption (and detection) of a single host leads to the exclusion of a domain, and with more hosts in each domain, chances of corruption of a domain are higher.

Figure 5.2(e) shows that the number of replicas running for an application decreases slightly with an increase in the number of hosts per domain. This is due to the decrease in the number of available good domains in which new replicas can be started. Figure 5.2(f) indicates that the load on each host, as represented by the number of application replicas (for all applications) running on the host, decreases significantly with an increase in the number of hosts per domain.

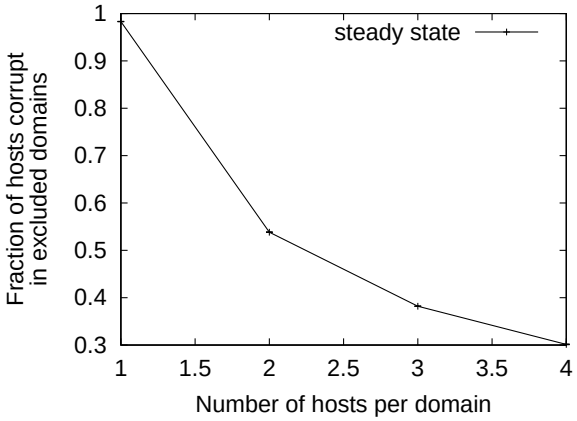
From the above two studies, we observe that it is clearly better to put as few hosts per



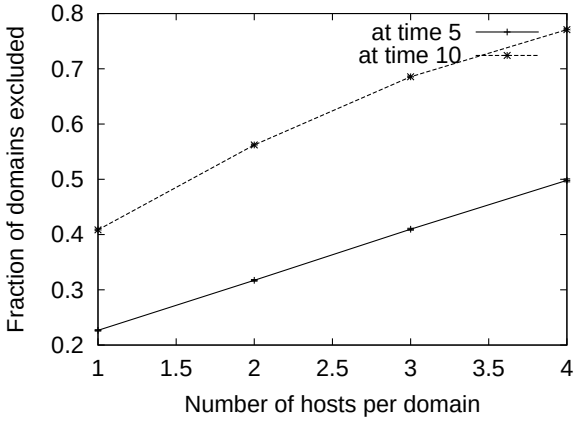
(a) Unavailability



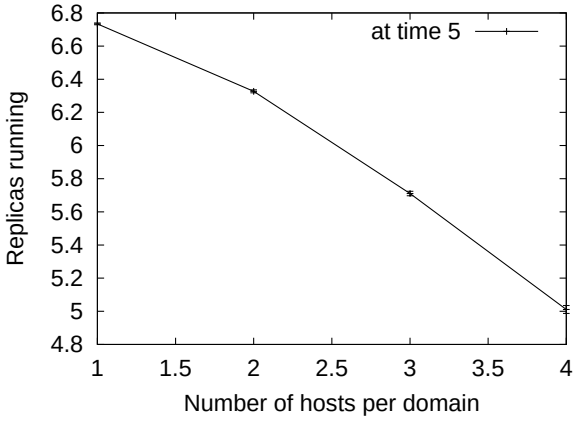
(b) Unreliability



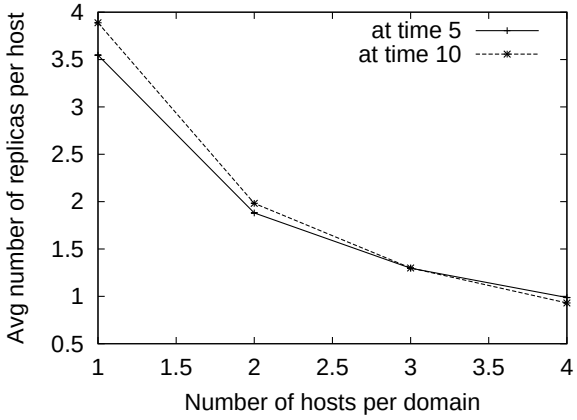
(c) Fraction of Corrupt Hosts in an Excluded Domain



(d) Fraction of Domains Excluded at the End of 5 Hours



(e) Number of Replicas of an Application After 5 Hours



(f) Number of Replicas on a Host After 5 Hours

Figure 5.2: Variation in Measures for Different Numbers of Hosts in 10 Domains

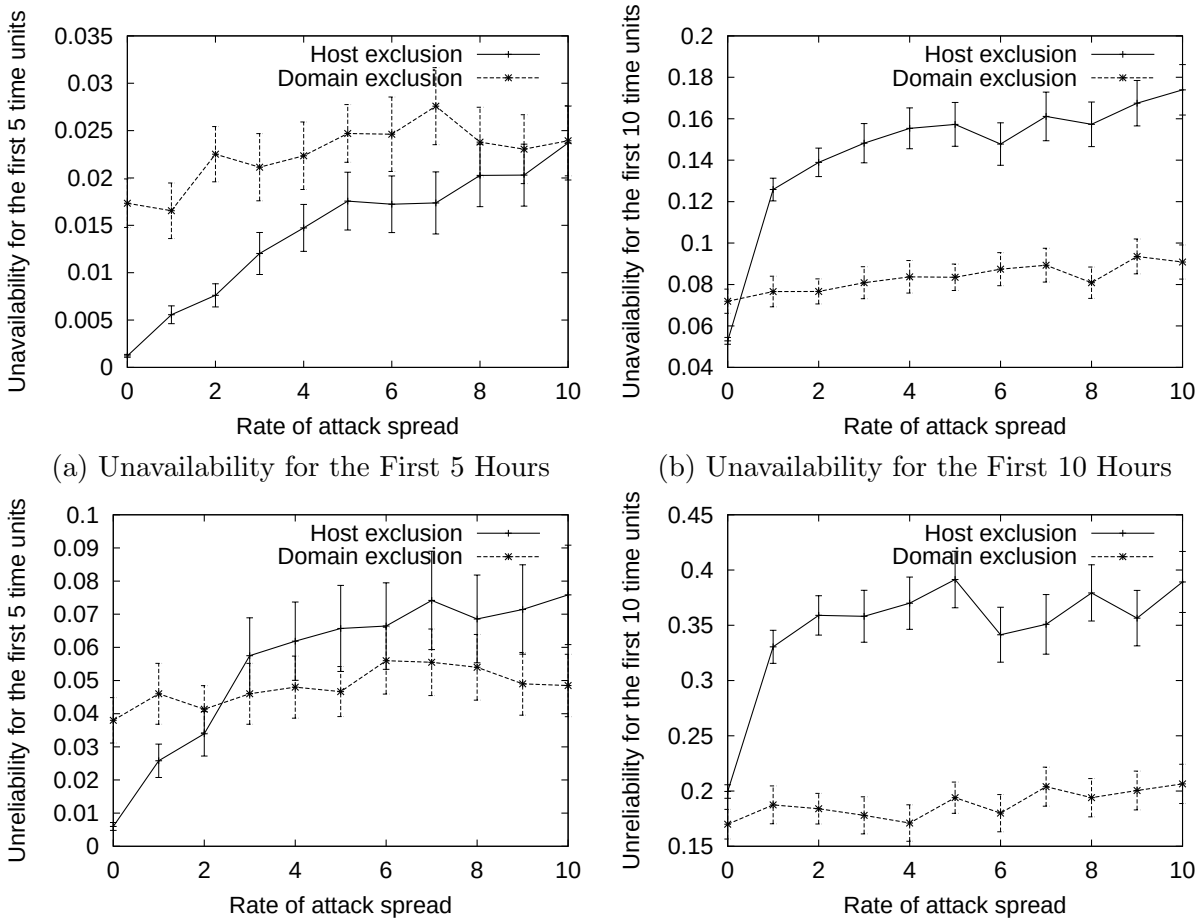
domain as possible. The second study indicates that increasing the number of hosts per domain does not provide any significant improvement, even when the number of hosts in the system (and hence cost) increases significantly. Hence, our studies suggest that unless constrained by physical limitations (such as those that might be caused by network design or firewall placement), it is advisable to form more domains by having fewer hosts per domain.

5.3 Comparison of Domain-exclusion and Host-exclusion Management Algorithms

Another major management issue is that of what to exclude when a host (or a replica on it) is found to be corrupt (assuming multiple hosts per domain). One approach is to exclude the entire domain that contains the host. This is a preemptive strike against the attackers, using the assumption that the attack may have spread to other hosts in the domain. The other approach is to exclude only the detected host, thus saving resources. We designed experiments to study which of the approaches was better for different rates of attack spread.

In the following set of experiments, we assumed that the corruption of the host operating system and services increased fivefold the chances that the replicas and management entity running on the host would be corrupt. The parameter values for the experiment were the same as for the previous experiments, except that we had 10 domains with 3 hosts per domain, and 4 applications with 7 replicas each. The within-domain attack spread rate varied from 0 (low) to 10 (high). We would like to remind the reader that the spread rate determines how quickly the attack on a host affects the other hosts in its domain, as well as how much the attack increases the vulnerability of other hosts. Before a corrupt host is detected and excluded from the group communication system, it may have spread the attack to multiple other hosts in the domain: that is the motivation for preemptive domain exclusion. A spread rate of 5 or more is quite high, but may be reasonable for the scenario considered, since major hardening is done at inter-domain boundaries, and not so much within domains.

Figure 5.3(a) shows that in the short run (5 hours) for low values of attack spread, exclusion of a single host provides better application availability than the domain exclusion scheme does. However, the two perform similarly for high values of attack spread rate. However, as shown in Figure 5.3(b), the domain-exclusion scheme outperforms the host-exclusion scheme in the longer run (10 hours) for most values of the attack spread rate. As expected, the attack spread rate does not have much effect on the performance of the domain-exclusion scheme.



(a) Unavailability for the First 5 Hours

(b) Unavailability for the First 10 Hours

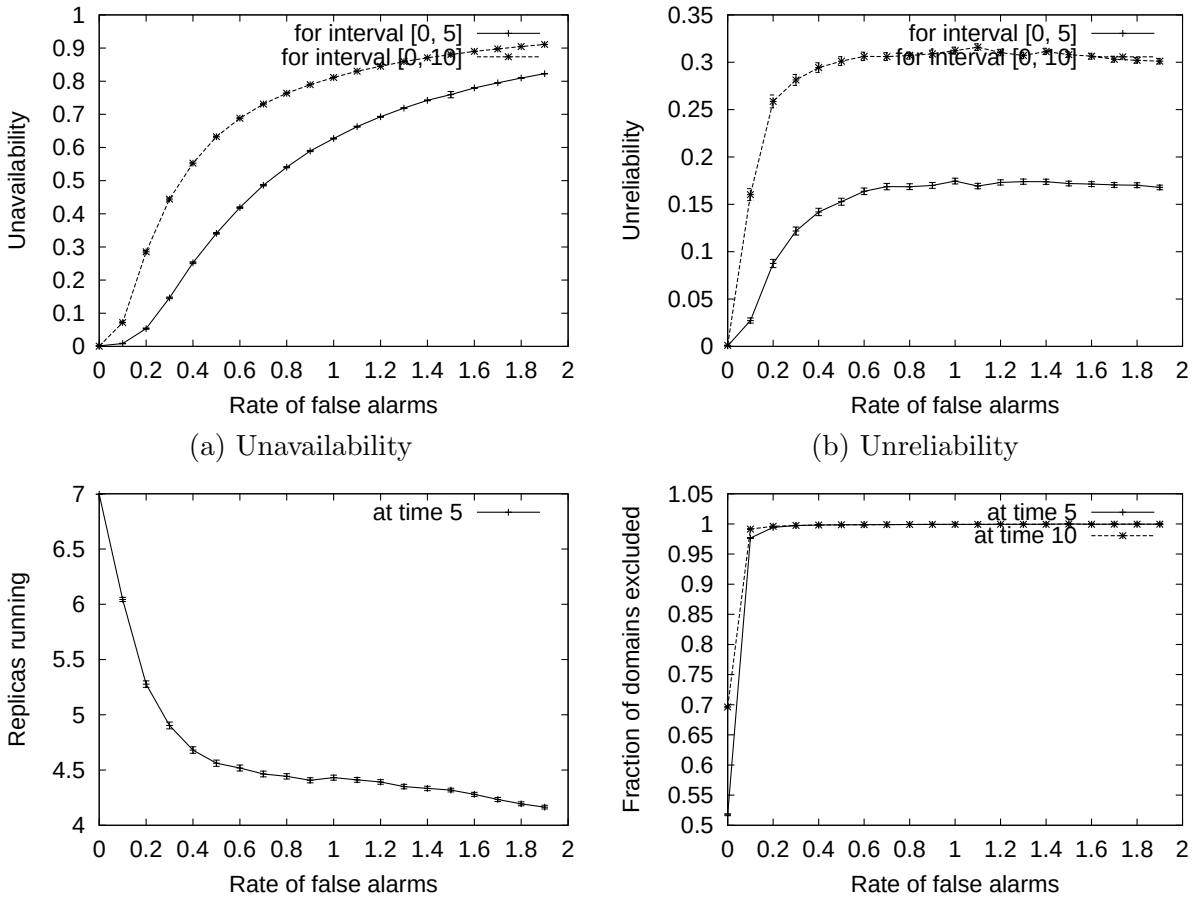
(c) Unreliability for the First 5 Hours

(d) Unreliability for the First 10 Hours

Figure 5.3: Unavailability and Unreliability for Different Exclusion Algorithms

Figure 5.3(c) shows that under the domain-exclusion scheme, application reliability does not change much as the within-domain attack spread rate changes, but that it is sensitive to the spread rate under the host-exclusion scheme. For the parameter ranges studied, the domain-exclusion scheme provides better application reliability for spread rates of 4 or more (for the first 5 hours). Figure 5.3(d) shows that the domain-exclusion scheme outperforms the host-exclusion scheme for almost all spread rate values for the longer time run of 10 hours.

The above results indicate that for the studied attack and detection rates, even for a somewhat high within-domain attack spread rate, a preemptive-action-based domain-exclusion scheme performs almost as well as the host-exclusion scheme in the short run, and significantly better in the long run.



(a) Unavailability

(b) Unreliability

(c) Number of Replicas of an Application after 5 Hours

(d) Fraction of Domains Excluded after 5 Hours

Figure 5.4: Variation in Measures for Different Rates of False Alarms

5.4 Impact of Quality of Intrusion Detection

We also conducted a study to analyze the effect on system performance of the quality of the intrusion detection software. Specifically, we studied the effect of the rate of false alarms. The parameters for the experiments were similar to those used in the previous experiments, except that the rate of false alarms generated by the intrusion detection software on each host was varied from 0 to 2.0. Note that the high end of this range of values is rather extreme and would not be experienced by typical intrusion detection systems (the cumulative system-wide false alarm rate is about 100 times the per-host rate). We consider 10 domains with 4 hosts each, and 4 applications with 7 replicas each.

Figures 5.4(a) and 5.4(b) show how application unavailability and unreliability changed the rate of false alarms. The system performance deteriorates rapidly and then stabilizes, since most domains are excluded after a certain point, as shown by Figures 5.4(c) and 5.4(d).

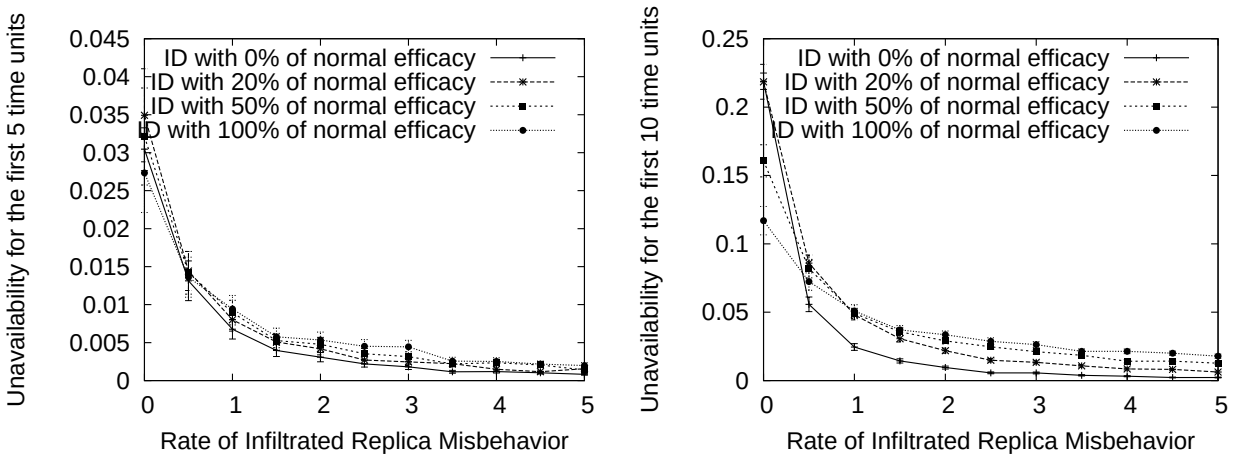


Figure 5.5: Unavailability for Different Misbehavior Rates of Infiltrated Replicas (First Study)

As the results indicate, it is very important to control the false alarm rate, even if that means reducing the valid detection rate.

5.5 Effect of the Rate of Misbehavior by Infiltrated Replicas

We conducted experiments on the model to study the effect of the rate at which the infiltrated replicas exhibit *corrupt behavior*. As mentioned in Section 3 and 4, corrupt behavior refers to incorrect group-communication-system-level behavior exhibited by an infiltrated replica that is detected by the good replicas in its replication group, hence making the good replicas aware of the infiltrated replica's corruption. That is one of the two ways in which intrusions are detected in the system, the other being intrusion detection software installed in the hosts.

We studied the variation in system performance (intrusion tolerance) with varying misbehavior rates, and also while simultaneously changing the quality of the intrusion detection software. The efficacy of the intrusion detection system is specified in the model primarily through the probabilities of detection of various kinds of attacks. For our experiments, we defined the following probabilities of detection as "normal," i.e., as a base case with values typical of a modern IDS:

- Probability of detection of script-based attacks on host OS and services: 95%
- Probability of detection of more exploratory attacks on host OS and services: 75%

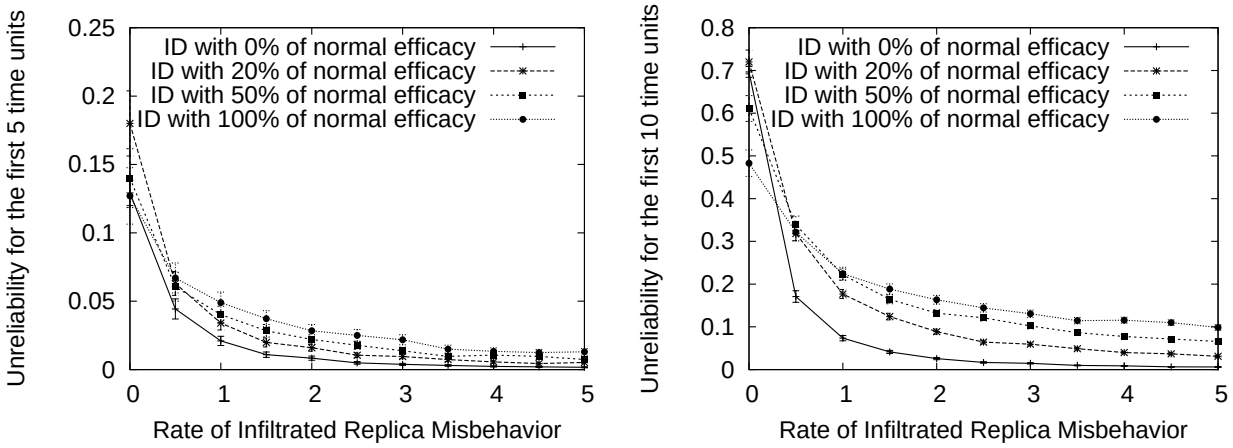


Figure 5.6: Unreliability for Different Misbehavior Rates of Infiltrated Replicas (First Study)

- Probability of detection of innovative attacks on host OS and services: 40%
- Probability of detection of attacks on management entities: 80%
- Probability of detection of attacks on application replicas: 80%

We then studied how the system's intrusion tolerance varied with the misbehavior rate of infiltrated replicas for four sets of detection probabilities of the intrusion detection software: no detection at all, 20% of "normal" (all detection probabilities cut to 20% of the values defined above), 50% of "normal," and the "normal" set of values defined above. The cumulative base attack rate on the system was set to be 5 successful attacks per time unit, with 3 of these being attacks on host OS and services, and 1 being direct attacks on management entities, and 1 being direct attacks on application replicas. We considered a system with 10 security domains with 2 hosts each, and 4 applications with 7 replicas each. The infiltration of a host OS and services increased the chances of infiltration of replicas on the host five-fold. The remaining parameters were the same as those used for experiments described earlier in this section.

Figures 5.5 and 5.6 show how system unavailability and unreliability change with the misbehavior rate of infiltrated replicas. From the wide range of values for the measures, it is clear that this parameter has a significant impact on the level of intrusion tolerance offered by the system.

Several interesting results come out when we also vary the level of intrusion detection. Systems with better-quality intrusion detection software perform better when the misbehavior rate is low (e.g., observe the y-intercept in the graphs of Figures 5.5 and 5.6, i.e., no misbehavior) and ID software is the primary means of detection of infiltration. However,

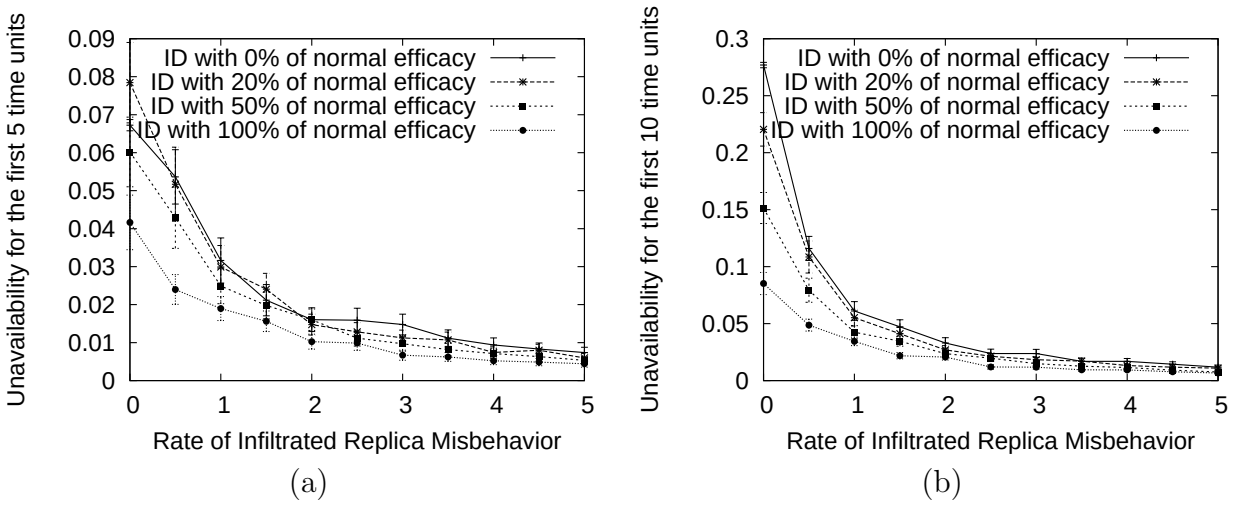


Figure 5.7: Unavailability for Different Misbehavior Rates of Infiltrated Replicas (Second Study)

as the misbehavior rate improves, the systems with lower-quality ID software outperform the systems with better software. While this seems counterintuitive, it can be explained as follows. For the present set of experiments, the majority of the contribution to the base attack rate comes from the attacks on the host's OS and services. While infiltration of a host increases the chances of replicas on the host being infiltrated too, if the ID software is good, it can detect the intrusion before it has a chance to corrupt the replicas on the host. This can result in the exclusion of the security domain with the infiltrated host, though the replicas in the domain are still good. It results in exhaustion of security domains, decreasing the possible level of redundancy and adversely affecting the application availability. In the longer run (10 time units), more domains have been excluded; hence, the playing field is more balanced, and the minimum misbehavior rate for which systems with lower-quality ID are able to outperform systems with higher-quality ID increases.

To further corroborate our explanation, we designed a second study with the following changes:

- Infiltration of a host increased the chances of corruption of replicas running on the host 1000 times (instead of 5 times).
- Distribution: 20 domains with 1 host each (instead of 10 domains with 2 hosts each).
- Cumulative base attack rate of 5 per time unit includes of 2 attacks on application processes, 2 attacks on host OS and services, and 1 attack on management entities (instead of a 3:1:1 distribution).

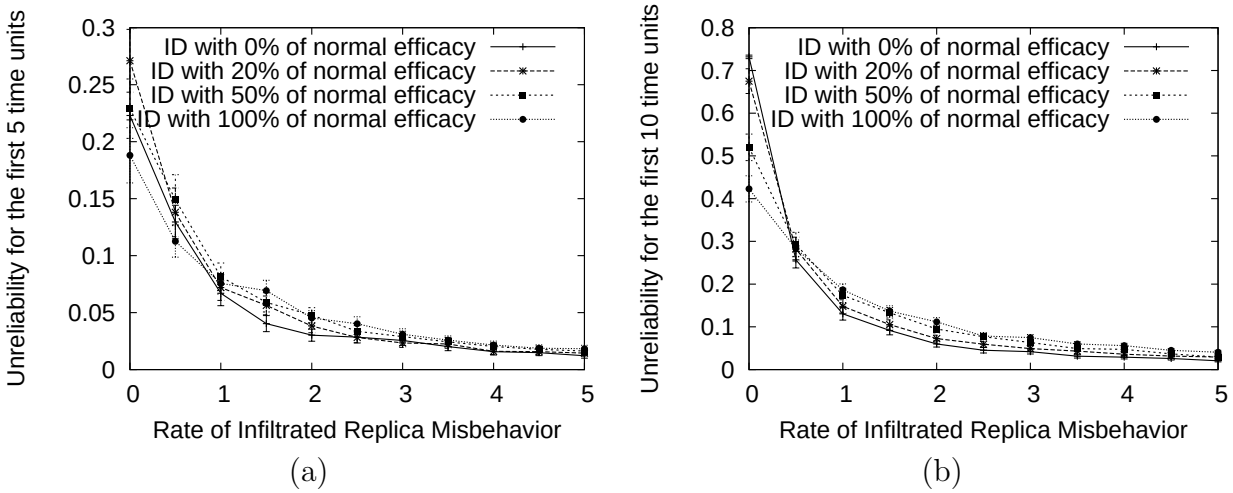


Figure 5.8: Unreliability for Different Misbehavior Rates of Infiltrated Replicas (Second Study)

Figures 5.7 and 5.8 show the variation in system unavailability and unreliability for the experiments in this study. Since now the attack spreads quickly to the replica processes on infiltrated hosts, and the system has more domains, the configuration with better-quality intrusion detection software performs better than configurations with poorer-quality software. That substantiates our reasoning for the results from the first set of experiments in this section.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we presented a probabilistic validation of an intrusion-tolerant replication system. The results are significant for the following reasons. First, they demonstrate the utility of probabilistic modeling for validating complex intrusion-tolerant architectures, and show that stochastic activity networks are an appropriate model representation for this purpose. A model abstracts a system's implementation and behavior. Models can be used for validation because it is easier to analyze properties of a model, such as integrity or availability, than it is to analyze the same properties of the real system. The SAN model we created was designed to be modular, so that it could be easily modified to represent other intrusion-tolerant systems.

Second, the results present useful insights into the relative merits of various design choices for the ITUA replication management system. The results show that for the ITUA replication management system, it was advisable to put as few hosts per domain as the physical constraints would allow, since the intrusion tolerance offered by the system was highly sensitive to the number of security domains available for starting new replicas. We also showed that it was important to limit the rate of false alarms, even if the rate of valid detection had to be sacrificed a little to achieve that. We studied another management scheme in which only the corrupt host is excluded, and observed that if an attack can spread quickly within a domain, it is better to exclude the entire domain when an intrusion is detected. We showed that the system is quite sensitive to the delay between the corruption of a replica and the actual exhibition of corrupt behavior by the replica during group communication. If corrupted replicas lie dormant for a long time, a large number of corrupted replicas in a replication group tend to misbehave almost simultaneously, resulting in Byzantine failure. However, such behavior is a complex function of the efficacy of the intrusion detection system

and the ease of spread of attack within a security domain.

6.2 Future Work

A logical next step to the present work is to apply this approach to several other intrusion-tolerant systems, to validate their architectures and aid in their design processes. There is a need to explore ways to combine the probabilistic modeling with a mechanism for generating experimental data to obtain accurate values for various input parameters.

Probabilistic modeling as described in this thesis can also be used as a part of an integrated validation procedure for validating survivable systems against a given set of quantitative requirements. Such a procedure can use a wide array of validation techniques, which can be used both to process output from models to produce higher-level abstractions and to as feed values into the models.

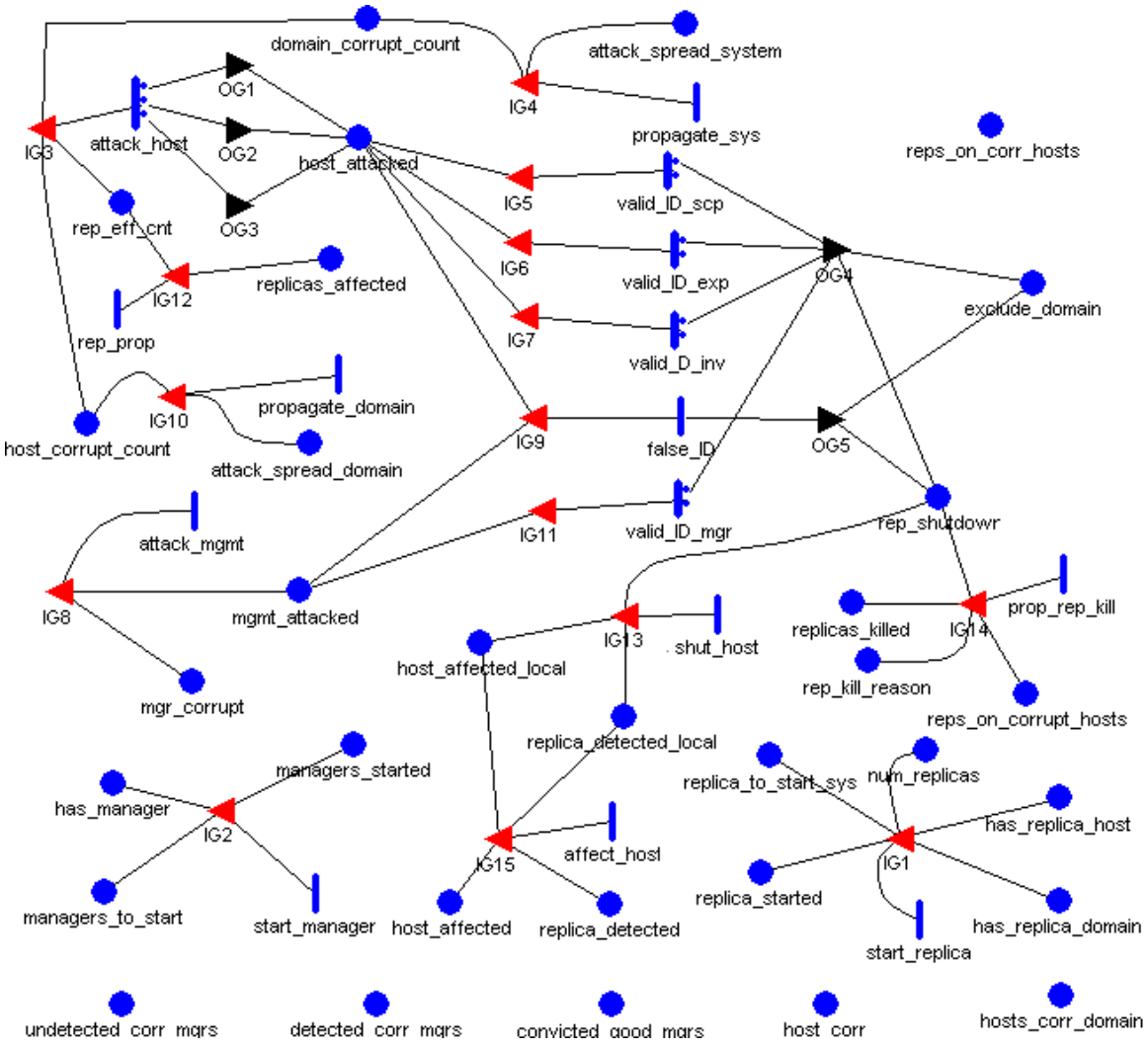
The long-term goal of this research will be to develop a validation framework, which would use probabilistic modeling as a component, and could be quickly and automatically applied to validate generic security mechanisms.

Appendix A

Model Documentation

The following is the documentation for the SAN model, generated by Möbius.

Model: Host_module



Place Names	Initial Markings
attack_spread_domain	0
attack_spread_system	0
convicted_good_mgrs	0
detected_corr_mgrs	0
domain_corrupt_count	0
exclude_domain	0
has_manager	0
has_replica_domain	0
has_replica_host	0
host_affected	0
host_affected_local	0
host_attacked	0
host_corr	0
host_corrupt_count	0
hosts_corr_domain	0
managers_started	0
managers_to_start	1
mgmt_attacked	0
mgr_corrupt	0
num_replicas	0
rep_eff_cnt	0
rep_kill_reason	0
rep_shutdown	0
replica_detected	0
replica_detected_local	0
replica_started	0
replica_to_start_sys	0
replicas_affected	0
replicas_killed	0
reps_on_corr_hosts	0
reps_on_corrupt_hosts	0
undetected_corr_mgrs	0

Timed Activity:	affect_host
Exponential Distribution Parameters	Rate `host_affect_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	attack_host
Exponential Distribution Parameters	Rate `base_host_attack_rate * (attack_spread_domain->Mark() + 1.0) * (attack_spread_system->Mark()+1.0)`

Activation Predicate	`1`
Reactivation Predicate	`1`

Case Distributions	**case 1**
	`prob_script_attack`
	case 2
	`1-(prob_script_attack+prob_innovative_attack)`
	case 3
	`prob_innovative_attack`

Timed Activity:	**attack_mgmt**

Exponential Distribution Parameters	**Rate**
	`base_mgmt_attack_rate*`
	`((2*host_attacked->Mark())+1.0)*`
	`(attack_spread_domain->Mark()+1.0)*`
	`(attack_spread_system->Mark()+1.0)`

Activation Predicate	`1`
Reactivation Predicate	`1`

Timed Activity:	**false_ID**
Exponential Distribution Parameters	**Rate** `false_alarm_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**prop_rep_kill**
Exponential Distribution Parameters	**Rate** `rep_kill_prop_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**propagate_domain**
Exponential Distribution Parameters	**Rate** `domain_prop_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**propagate_sys**
Exponential Distribution Parameters	**Rate** `sys_prop_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**rep_prop**

Exponential Distribution Parameters	**Rate** `rep_eff_prop_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**shut_host**
Exponential Distribution Parameters	**Rate** `host_shutdown_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**start_manager**
Exponential Distribution Parameters	**Rate** `mgr_start_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**start_replica**
Exponential Distribution Parameters	**Rate** `rep_start_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**valid_D_inv**
Exponential Distribution Parameters	**Rate** `succ_inv_ID_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Case Distributions	**case 1**
	
	`prob_ID_inv`
	case 2
	
	`1-prob_ID_inv`

Timed Activity:	**valid_ID_exp**
Exponential Distribution Parameters	**Rate** `succ_exp_ID_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Case Distributions	**case 1**
	
	`prob_ID_exp`
	case 2
	
	`1-prob_ID_exp`

Timed Activity:	**valid_ID_mgr**

Exponential Distribution Parameters	**Rate** succ_mgr_ID_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Case Distributions	**case 1** prob_ID_mgr **case 2** 1-prob_ID_mgr
Timed Activity:	**valid_ID_scp**
Exponential Distribution Parameters	**Rate** succ_scp_ID_rate
Activation Predicate	(none)
Reactivation Predicate	(none)
Case Distributions	**case 1** prob_ID_scp **case 2** 1-prob_ID_scp

Input Gate:	**IG1**
Predicate	/* If the system has a replica to start, the domain is * not already excluded, and the domain already doesn't * have a replica of this app (bitwise and) */ replica_to_start_sys->Mark() && exclude_domain->Mark()==0 && ((~((unsigned short)has_replica_domain->Mark())) & (unsigned short)replica_to_start_sys->Mark()) && replica_started->Mark()==0

Function	/* Start the replica, updating info for host and domain,
	and
	* reset replica_to_start_sys
	*/
	unsigned short a =
	~((unsigned short)has_replica_domain->Mark());
	replica_started->Mark() =
	(short)(a & (unsigned
	short)replica_to_start_sys->Mark());
	has_replica_host->Mark() =
	has_replica_host->Mark() | replica_started-
	>Mark();
	has_replica_domain->Mark() =
	has_replica_domain->Mark() | replica_started-
	>Mark();
	replica_to_start_sys->Mark() -= replica_started->Mark();
	
	/* find the number of 1's in replica_started */

	int rep = replica_started->Mark();
	int sum=0;
	while(rep!=0) {
	int mod = rep%2;
	rep = rep/2;
	sum += mod;
	}
	num_replicas->Mark() += sum;

Input Gate:	**IG10**
Predicate	host_corrupt_count->Mark()==1
Function	attack_spread_domain->Mark()++; host_corrupt_count->Mark() = 2;
Input Gate:	**IG11**
Predicate	mgmt_attacked->Mark() && exclude_domain->Mark()==0
Function	;
Input Gate:	**IG12**
Predicate	rep_eff_cnt->Mark()==1 && replicas_affected->Mark() == 0 && exclude_domain->Mark()==0
Function	rep_eff_cnt->Mark() = 0; replicas_affected->Mark() = has_replica_host->Mark();
Input Gate:	**IG13**
Predicate	replica_detected_local->Mark() && host_affected_local->Mark() && exclude_domain->Mark()==0 && rep_shutdown->Mark()==0

Function	if(host_corr->Mark()==0) {
	hosts_corr_domain->Mark()++;
	host_corr->Mark()=1;
	}
	
	/* similar to OG4: if manager and subordinate ok, or enough
	* good managers to convict this one */
	if((mgmt_attacked->Mark()==0 && mgr_corrupt->Mark()==0)
	|| (num_domains - (convicted_good_mgrs->Mark()
	+
	detected_corr_mgrs->Mark())
	> 3*undetected_corr_mgrs->Mark()))
	{
	rep_shutdown->Mark() = 2;
	host_affected_local->Mark() = 0;
	exclude_domain->Mark() = 1;
	if(mgr_corrupt->Mark()) {
	undetected_corr_mgrs->Mark()--;

	}
	detected_corr_mgrs->Mark()++;
	}

Input Gate:	**IG14**
Predicate	/* replicas need to be shutdown, as the domain has been * excluded. All surrounding places are shared at domain * level. */ rep_shutdown->Mark() && replicas_killed->Mark() == 0 && has_replica_domain->Mark()

Function	replicas_killed->Mark() = has_replica_domain->Mark();
	if(rep_shutdown->Mark()==1) {
	rep_kill_reason->Mark()=0;
	}
	else {
	rep_kill_reason->Mark() =
	replica_detected_local->Mark();
	replica_detected_local->Mark()=0;
	}
	rep_shutdown->Mark() = 0;
	has_replica_domain->Mark()=0;
	reps_on_corrupt_hosts->Mark() = reps_on_corr_hosts-
	>Mark();

Input Gate:	**IG15**
Predicate	/* A corrupt replica has been detected, and we need to * exclude the relevant host/domain. Choose appropriate * host depending on whether it has this replica and it is * corrupted if host_affected is true (i.e. corrupt replica * was on a corrupt host */ (replica_detected->Mark() & has_replica_host->Mark()) && host_affected->Mark() == host_attacked->Mark() && exclude_domain->Mark()==0 && rep_shutdown->Mark()==0 && (replica_detected_local->Mark() & replica_detected- >Mark())==0

Function	replica_detected_local->Mark() |= replica_detected-
	>Mark();
	host_affected_local->Mark() = host_affected->Mark();
	replica_detected->Mark()=0;
	host_affected->Mark()=0;

Input Gate:	**IG2**
Predicate	/* There is no manager in the domain currently */ managers_to_start->Mark()
Function	

	managers_to_start->Mark() = 0;
	has_manager->Mark() = 1;
	managers_started->Mark()++;

Input Gate:	**IG3**
Predicate	/* Host is not already corrupt */ host_attacked->Mark()==0

Function	if(host_corrupt_count->Mark()==0) {
	host_corrupt_count->Mark() = 1;
	}
	if(domain_corrupt_count->Mark()==0) {
	domain_corrupt_count->Mark() = 1;
	}
	if(rep_eff_cnt->Mark()==0) {
	rep_eff_cnt->Mark() = 1;
	}
	if(exclude_domain->Mark()==0 && host_corr->Mark()==0) {
	hosts_corr_domain->Mark()++;
	host_corr->Mark()=1;
	}
	reps_on_corr_hosts->Mark() |= has_replica_host->Mark();

Input Gate:	**IG4**
Predicate	/* check if propagation effect due to attack on this host * has already been accounted for, and if the domain's not * already excluded */ domain_corrupt_count->Mark()==1 && exclude_domain->Mark()==0

Function	attack_spread_system->Mark()++;
	
	/* indicate that we have counted this host */
	domain_corrupt_count->Mark()=2;

Input Gate:	**IG5**
Predicate	host_attacked->Mark()==1 && mgmt_attacked->Mark()==0 && exclude_domain->Mark()==0
Function	;
Input Gate:	**IG6**
Predicate	host_attacked->Mark()==2 && mgmt_attacked->Mark()==0 && exclude_domain->Mark()==0
Function	;
Input Gate:	**IG7**

Predicate	`host_attacked->Mark()==3 && mgmt_attacked->Mark()==0 && exclude_domain->Mark()==0`
Function	`;`

Input Gate:	**IG8**
Predicate	`mgmt_attacked->Mark()==0`

Function	`/* if domain not already excluded and this is a manager,`
	` * updated state`
	` */`
	`if(exclude_domain->Mark()==0 && has_manager->Mark()) {`
	`  undetected_corr_mgrs->Mark()++;`
	`}`
	`if(domain_corrupt_count->Mark()==0) {`
	`  domain_corrupt_count->Mark()=1;`
	`}`
	`if(host_corrupt_count->Mark()==0) {`
	`  host_corrupt_count->Mark()=1;`
	`}`
	`if(has_manager->Mark()) {`
	`  mgr_corrupt->Mark() = 1;`
	`}`
	`if(exclude_domain->Mark()==0 && host_corr->Mark()==0) {`
	`  hosts_corr_domain->Mark()++;`
	`  host_corr->Mark()=1;`
	`}`

Input Gate:	**IG9**
Predicate	`host_attacked->Mark()==0 && mgmt_attacked->Mark()==0 && exclude_domain->Mark()==0`
Function	`;`

Output Gate:	**OG1**
Function	`host_attacked->Mark()=1;`

Output Gate:	**OG2**
Function	`host_attacked->Mark()=2;`

Output Gate:	**OG3**
Function	`host_attacked->Mark()=2;`

Output Gate:	**OG4**

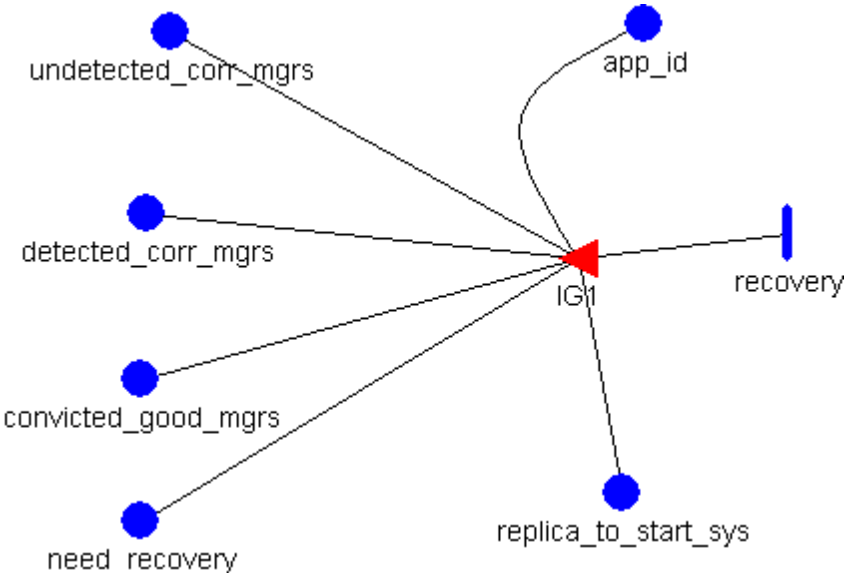
Function	`/* if the local subordinate is not corrupt and domain's`
	` * manager is not corrupt, detection by local loops would`
	` * cause the domain to exclude itself.`

	*/
	if(mgmt_attacked->Mark()==0 && mgr_corrupt->Mark()==0) {
	exclude_domain->Mark() = 1;
	if(has_replica_domain->Mark()) {
	rep_shutdown->Mark() = 1;
	}
	if(domain_corrupt_count->Mark()==2) {
	attack_spread_system->Mark()--;
	}
	detected_corr_mgrs->Mark()++;
	
	if((replica_detected_local->Mark() &
	has_replica_host->Mark()) &&
	host_affected_local->Mark() == host_attacked-
	>Mark()) {
	rep_shutdown->Mark()=2;
	host_affected_local->Mark()=0;
	}
	}

Output Gate:	**OG5**

Function	if(mgr_corrupt->Mark() == 0) {
	exclude_domain->Mark() = 1;
	rep_shutdown->Mark() = 1;
	convicted_good_mgrs->Mark()++;
	if(domain_corrupt_count->Mark()==2) {
	attack_spread_system->Mark()--;
	}
	if((replica_detected_local->Mark() &
	has_replica_host->Mark()) &&
	host_affected_local->Mark() == host_attacked-
	>Mark()) {
	rep_shutdown->Mark()=2;
	host_affected_local->Mark()=0;
	}
	}

Model: ManagementAlgo_module

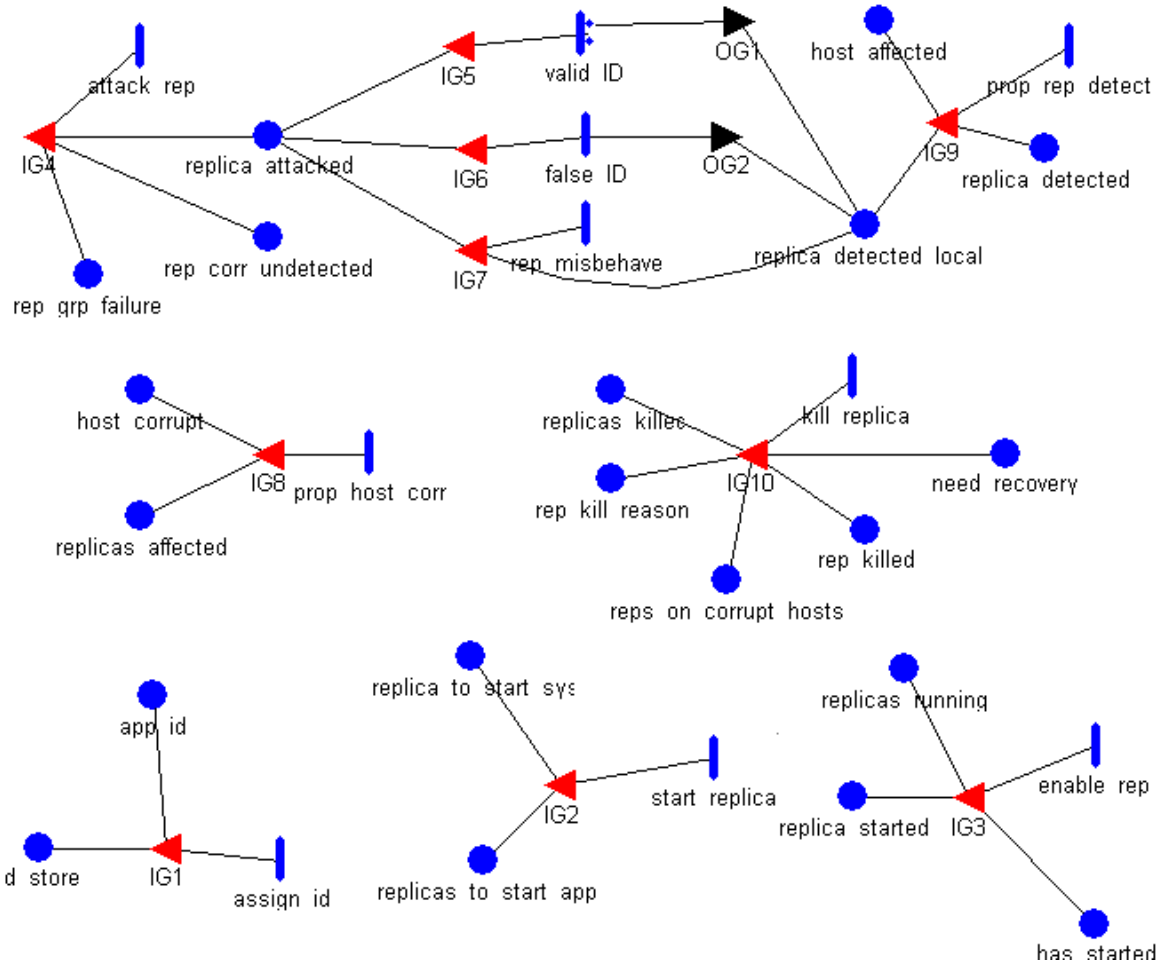


Place Names	Initial Markings
app_id	0
convicted_good_mgrs	0
detected_corr_mgrs	0
need_recovery	0
replica_to_start_sys	0
undetected_corr_mgrs	0

Timed Activity:	recovery
Exponential Distribution Parameters	Rate recovery_rate
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	IG1
Predicate	need_recovery->Mark() && (app_id->Mark() & replica_to_start_sys->Mark()) == 0 && (num_domains - (detected_corr_mgrs->Mark() + convicted_good_mgrs->Mark())) > 3*undetected_corr_mgrs->Mark()
Function	replica_to_start_sys->Mark() += app_id->Mark(); need_recovery->Mark()--;

Model: Replica_module



Place Names	Initial Markings
app_id	0
has_started	0
host_affected	0
host_corrupt	0
id_store	16384
need_recovery	0
rep_corr_undetected	0
rep_grp_failure	0
rep_kill_reason	0
rep_killed	0
replica_attacked	0
replica_detected	0
replica_detected_local	0
replica_started	0
replica_to_start_sys	0

replicas_affected	0
replicas_killed	0
replicas_running	0
replicas_to_start_app	num_reps
reps_on_corrupt_hosts	0

Timed Activity:	**assign_id**
Exponential Distribution Parameters	**Rate** `id_assign_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**attack_rep**
Exponential Distribution Parameters	**Rate** `base_rep_attack_rate*((host_corrupt->Mark()*1000)+1.0)`
Activation Predicate	`1`
Reactivation Predicate	`1`

Timed Activity:	**enable_rep**
Exponential Distribution Parameters	**Rate** `rep_enable_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**false_ID**
Exponential Distribution Parameters	**Rate** `false_alarm_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**kill_replica**
Exponential Distribution Parameters	**Rate** `rep_kill_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**prop_host_corr**
Exponential Distribution Parameters	**Rate** `host_corr_prop_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**prop_rep_detect**
Exponential Distribution Parameters	**Rate** `rep_detect_prop_rate`

Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**rep_misbehave**
Exponential Distribution Parameters	**Rate** `rep_misbehave_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**start_replica**
Exponential Distribution Parameters	**Rate** `replica_start_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)
Timed Activity:	**valid_ID**
Exponential Distribution Parameters	**Rate** `succ_rep_ID_rate`
Activation Predicate	(none)
Reactivation Predicate	(none)

Case Distributions	**case 1**
	
	`prob_ID_replica`
	case 2
	
	`1-prob_ID_replica`

Input Gate:	**IG1**
Predicate	`id_store->Mark() >= 1 &&` `app_id->Mark() == 0`
Function	`app_id->Mark() = id_store->Mark();` `id_store->Mark() = id_store->Mark()/2;`
Input Gate:	**IG10**

Predicate	`has_started->Mark() &&`
	`(replicas_killed->Mark() & app_id->Mark()) &&`
	`(((rep_kill_reason->Mark() & app_id->Mark()) &&`
	`replica_detected_local->Mark()==2) ||`
	`((rep_kill_reason->Mark() & app_id->Mark())==0 &&`
	`replica_detected_local->Mark()!=2 &&`
	`( ((reps_on_corrupt_hosts->Mark() & app_id->Mark()) &&`
	`host_corrupt->Mark()) ||`
	`((reps_on_corrupt_hosts->Mark() & app_id->Mark())==0 &&`
	`host_corrupt->Mark()==0))))`

Function	`rep_killed->Mark() = 1;`
	`need_recovery->Mark() ++;`
	`replicas_running->Mark()--;`

	replicas_killed->Mark() -= app_id->Mark();
	if(replicas_killed->Mark() == 0) {
	reps_on_corrupt_hosts->Mark()=0;
	}
	/* and some other stuff if this has to be reused
	* -- reinitialize various places
	*/
	replica_attacked->Mark()=0;
	replica_detected_local->Mark() = 0;
	has_started->Mark()=0;

Input Gate:	**IG2**
Predicate	/* Application has been assigned an ID, there * are replicas to start, and there is already * not a replica in the system buffer */ app_id->Mark() && replicas_to_start_app->Mark() && (replica_to_start_sys->Mark() & app_id->Mark()) == 0
Function	/* Decrease the number of replicas remaining to be started. * Put in the system buffer info abt replica for which * application needs to started next. */ replicas_to_start_app->Mark()--; replica_to_start_sys->Mark() = replica_to_start_sys->Mark() | app_id->Mark();
Input Gate:	**IG3**
Predicate	has_started->Mark() == 0 && (replica_started->Mark() & app_id->Mark())

Function	has_started->Mark() = 1;
	replica_started->Mark() -= app_id->Mark();
	replicas_running->Mark()++;

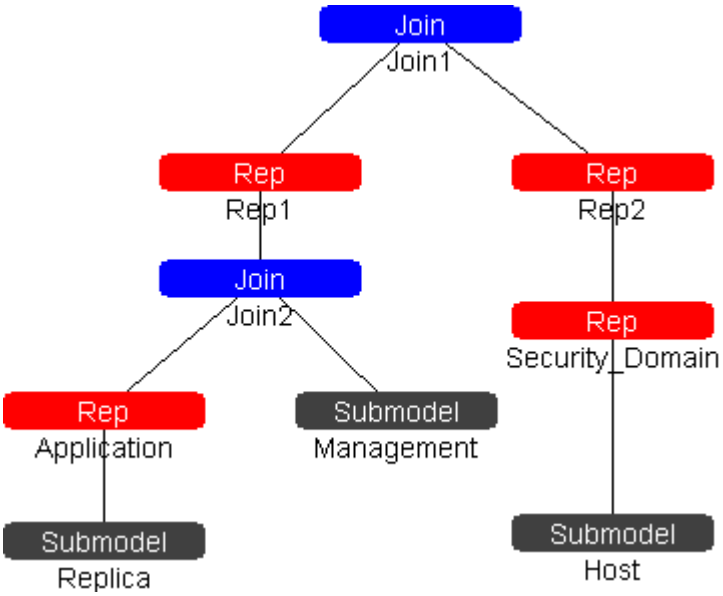
Input Gate:	**IG4**
Predicate	replica_attacked->Mark() == 0 && has_started->Mark()

Function	replica_attacked->Mark() = 1;
	rep_corr_undetected->Mark()++;
	if((replicas_running->Mark()) <= 3*rep_corr_undetected->Mark()) {
	rep_grp_failure->Mark() = 1;
	}

Input Gate:	**IG5**
Predicate	replica_detected_local->Mark()==0 && replica_attacked->Mark()
Function	;

Input Gate:	IG6
Predicate	`replica_attacked->Mark() == 0 && replica_detected_local->Mark() ==0`
Function	`;`
Input Gate:	**IG7**
Predicate	`replica_attacked->Mark() && replica_detected_local->Mark() == 0 && (replicas_running->Mark()) > 3*rep_corr_undetected->Mark()`
Function	`replica_detected_local->Mark() = 1; rep_corr_undetected->Mark()--;`
Input Gate:	**IG8**
Predicate	`(replicas_affected->Mark() & app_id->Mark()) && host_corrupt->Mark() == 0`
Function	`host_corrupt->Mark() = 1; replicas_affected->Mark() -= app_id->Mark();`
Input Gate:	**IG9**
Predicate	`replica_detected_local->Mark()==1 && replica_detected->Mark() == 0`
Function	`replica_detected_local->Mark() == 2; replica_detected->Mark() = app_id->Mark(); host_affected->Mark() = host_corrupt->Mark();`
Output Gate:	**OG1**
Function	`replica_detected_local->Mark()=1; rep_corr_undetected->Mark()--;`
Output Gate:	**OG2**
Function	`replica_detected_local->Mark() = 1;`

Model: ITUA_model



Rep Node	Reps	Shared State Variables
Application	num_reps	app_id
		host_affected
		id_store
		need_recovery
		rep_corr_undetected
		rep_grp_failure
		rep_kill_reason
		replica_detected
		replica_started
		replica_to_start_sys
		replicas_affected
		replicas_killed
		replicas_running
		replicas_to_start_app
		reps_on_corrupt_hosts
Rep1	num_apps	convicted_good_mgrs
		detected_corr_mgrs
		host_affected
		id_store
		rep_kill_reason
		replica_detected
		replica_started
		replica_to_start_sys
		replicas_affected
		replicas_killed

		reps_on_corrupt_hosts
		undetected_corr_mgrs
Rep2	num_domains	attack_spread_system
		convicted_good_mgrs
		detected_corr_mgrs
		host_affected
		managers_started
		rep_kill_reason
		replica_detected
		replica_started
		replica_to_start_sys
		replicas_affected
		replicas_killed
		reps_on_corrupt_hosts
		undetected_corr_mgrs
Security_Domain	num_hosts	attack_spread_domain
		attack_spread_system
		convicted_good_mgrs
		detected_corr_mgrs
		domain_corrupt_count
		exclude_domain
		has_replica_domain
		host_affected
		host_affected_local
		hosts_corr_domain
		managers_started
		managers_to_start
		mgr_corrupt
		rep_kill_reason
		rep_shutdown
		replica_detected
		replica_detected_local
		replica_started

Rep Node	Reps	Shared State Variables
Security_Domain	num_hosts	replica_to_start_sys
		replicas_affected
		replicas_killed
		reps_on_corr_hosts
		reps_on_corrupt_hosts
		undetected_corr_mgrs

Join Node: Join1 :

State Variable Name	Submodel Variables
convicted_good_mgrs	Rep2->convicted_good_mgrs

detected_corr_mgrs	Rep2->detected_corr_mgrs
host_affected	Rep1->host_affected
	Rep2->host_affected
rep_kill_reason	Rep1->rep_kill_reason
	Rep2->rep_kill_reason
replica_detected	Rep1->replica_detected
	Rep2->replica_detected
replica_started	Rep1->replica_started
	Rep2->replica_started
replica_to_start_sys	Rep1->replica_to_start_sys
	Rep2->replica_to_start_sys
replicas_affected	Rep1->replicas_affected
	Rep2->replicas_affected
replicas_killed	Rep1->replicas_killed
	Rep2->replicas_killed
reps_on_corrupt_hosts	Rep1->reps_on_corrupt_hosts
	Rep2->reps_on_corrupt_hosts
undetected_corr_mgrs	Rep2->undetected_corr_mgrs

Join Node: Join2 :

State Variable Name	Submodel Variables
app_id	Application->app_id
	Management->app_id
convicted_good_mgrs	Management->convicted_good_mgrs
detected_corr_mgrs	Management->detected_corr_mgrs
host_affected	Application->host_affected
id_store	Application->id_store
need_recovery	Application->need_recovery
	Management->need_recovery
rep_kill_reason	Application->rep_kill_reason
replica_detected	Application->replica_detected
replica_started	Application->replica_started
replica_to_start_sys	Application->replica_to_start_sys
	Management->replica_to_start_sys
replicas_affected	Application->replicas_affected
replicas_killed	Application->replicas_killed
reps_on_corrupt_hosts	Application->reps_on_corrupt_hosts
undetected_corr_mgrs	Management->undetected_corr_mgrs

Performance Variable Model: ITUA_PV

Child model name: ITUA_model
Model type: Rep/Join

Performance Variable : unavailability_5_id1			
Affecting Models	Replica		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `if(Replica->app_id->Mark()==16384 && ((Replica->replicas_running->Mark()) <= 3*Replica->rep_corr_undetected->Mark())) { return 1.0/(num_reps); }`		
Simulator Statistics	Type	Time Averaged 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	5.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.2

Performance Variable : unavailability_10_id2			
Affecting Models	Replica		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `if(Replica->app_id->Mark()==8192 && ((Replica->replicas_running->Mark()) <= 3*Replica->rep_corr_undetected->Mark())) { return 1.0/(num_reps); }`		
Simulator Statistics	Type	Time Averaged 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	10.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.2

Performance Variable : load_5

Affecting Models	Host		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `/* number of replicas on each host, averaged over all hosts * that have not been excluded yet */ if(Host->exclude_domain->Mark()==0) { return (Host->num_replicas->Mark()*1.0/ (num_hosts*(num_domains - Host->detected_corr_mgrs->Mark() - Host->convicted_good_mgrs->Mark()))); } else { return 0.0; }`		
Simulator Statistics	Type	Instant 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	5.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : num_replicas_5			
Affecting Models	Replica		
Impulse Functions			
Reward Function	*(Reward is over all Available Models)* `if(Replica->app_id->Mark()==16384) { return (Replica->replicas_running->Mark()*1.0/(num_reps)); }`		
Simulator Statistics	Type	Instant 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	5.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : load_10	
Affecting Models	Host
Impulse Functions	
Reward Function	*(Reward is over all Available Models)* `/* number of replicas on each host, averaged over all hosts * that have not been excluded yet */`

	`if(Host->exclude_domain->Mark()==0)`
	`{`
	`  return (Host->num_replicas->Mark()*1.0/`
	`  (num_hosts*(num_domains - Host->detected_corr_mgrs->Mark() -`
	`  Host->convicted_good_mgrs->Mark())));`
	`}`
	`else {`
	`  return 0.0;`
	`}`		

Simulator Statistics	Type	Instant 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	10.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : unreliability_5_id1			
Affecting Models	Replica		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Replica->app_id->Mark()==16384 &&`
	`  Replica->rep_grp_failure->Mark()) {`
	`  return 1.0/num_reps;`
	`}`		

Simulator Statistics	Type	Instant 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	5.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.2

Performance Variable : unreliability_10_id2			
Affecting Models	Replica		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Replica->app_id->Mark()==8192 && Replica->rep_grp_failure->Mark()) {`
	`  return 1.0/num_reps;`
	`}`		

Simulator Statistics	Type	Instant 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	10.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.2

Performance Variable : hosts_corrupt_100			
Affecting Models	Host		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Host->exclude_domain->Mark()==1) {`
	`  return (Host->host_corr->Mark()*1.0/`
	`  (Host->detected_corr_mgrs->Mark() +`
	`  Host->convicted_good_mgrs->Mark()));`
	`}`		

Simulator Statistics	Type	Instant 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	100.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : num_domains_5			
Affecting Models	Host		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`if(Host->exclude_domain->Mark()) {`
	`return Host->exclude_domain->Mark()*1.0/num_hosts;`
	`}`		

Simulator Statistics	Type	Instant 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	5.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : num_domains_10	
Affecting Models	Host
Impulse Functions	

Reward Function	*(Reward is over all Available Models)*
	`return ((Host->detected_corr_mgrs->Mark() +`
	`Host->convicted_good_mgrs->Mark())*1.0/`
	`(num_hosts*num_domains));`		

Simulator Statistics	Type	Instant 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	10.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

Performance Variable : ld_1			
Affecting Models	Host		
Impulse Functions			

Reward Function	*(Reward is over all Available Models)*
	`/* number of replicas on each host, averaged over all hosts`
	`* that have not been excluded yet`
	`*/`
	`if(Host->exclude_domain->Mark()==0)`
	`{`
	`return (Host->num_replicas->Mark()*1.0/`
	`(num_hosts*(num_domains - Host->detected_corr_mgrs->Mark() -`
	`Host->convicted_good_mgrs->Mark())));`
	`}`
	`else {`
	`return 0.0;`
	`}`		

Simulator Statistics	Type	Instant 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	1.0
	Confidence	Confidence Level	0.95
		Confidence Interval	0.1

References

[And72] 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.

[BD90] L. Blain and Y. Deswarte. Intrusion-Tolerant Security Server for Delta-4. In *Proceedings of the ESPRIT 90 Conference*, pages 355–370, November 1990.

[BDF02] R. V. Belani, S. M. Das, and D. Fisher. One-to-one Modeling and Simulation of Unbounded Systems: Experiences and Lessons. In *Proceedings of the 2002 Winter Simulation Conference*, pages 720–724, San Diego, CA, December 2002.

[CCD+01] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, and P. Webster. The Möbius Modeling Tool. In *Proceedings of the 9th Intl Workshop on Petri Nets and Performance Models*, pages 241–250, September 2001.

[CL99] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.

[CLR+02] T. Courtney, J. Lyons, H. V. Ramasamy, W. H. Sanders, M. Seri, M. Atighetchi, P. Rubel, C. Jones, F. Webber, P. Pal, R. Watro, M. Cukier, and J. Gossett. Providing Intrusion Tolerance with ITUA. In *Supplement of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, pages C–5–1– C–5–3, June 2002.

[DBF91] Y. Deswarte, L. Blain, and J. C. Fabre. Intrusion Tolerance in Distributed Computing Systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 110–121, May 1991.

[DCC+02] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius Framework and Its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, October 2002.

[DFF+88] Y. Deswarte, J. C. Fabre, J. M. Fray, D. Powell, and P. G. Ranea. Saturne: A Distributed Computing System which Tolerates Faults and Intrusions. In *Proceedings of the Workshop on Future Trends in Distributed Computing Systems in the 1990's*, pages 329–338, September 1988.

[EFL+97] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. Longstaff, and N. R. Mead. Survivable Network Systems: An Emerging Discipline. Technical Report CMU/SEI-97-TR-013, CMU Software Engineering Institute, November 1997.

[FDR94] J. C. Fabre, Y. Deswarte, and B. Randell. Designing Secure and Reliable Applications using Fragmentation-Redundancy-Scattering: An Object-Oriented Approach. In *Proceedings of the 1st European Dependable Computing Conference*, pages 21–38, October 1994.

[FP85] J. Fraga and D. Powell. A Fault and Intrusion-Tolerant File System. In *Proceedings of the IFIP 3rd International Conference on Computer Security*, pages 203–218, 1985.

[GGPW+01] F. Gong, K. Goševa-Popstojanova, F. Wang, R. Wang, K. Vaidyanathan, K. Trivedi, and B. Muthusamy. Characterizing Intrusion Tolerant Systems Using A State Transition Model. In *Proceedings of the DARPA Information Survivability Conference and Exposition II (DISCEX'01)*, pages 211–221, 2001.

[ISO99] 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).

[JO97] E. Jonsson and T. Olovsson. A Quantitative Model of the Security Intrusion Process Based on Attacker Behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, April 1997.

[JW01] S. Jha and J. M. Wing. Survivability Analysis of Networked Systems. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE2000)*, pages 307–317, 2001.

[Lan81] C. Landwehr. Formal Models for Computer Security. *Computer Surveys*, 13(3):247–278, September 1981.

[LBF+93] 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*, 2(2-3):211–229, 1993.

[Low01] J. Lowry. An Initial Foray into Understanding Adversary Planning and Courses of Action. In *Proceedings of the DARPA Information Survivability Conference and Exposition II (DISCEX'01)*, pages 123–133, 2001.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MGPVT02] B. B. Madan, K. Goševa-Popstojanova, K. Vaidyanathan, and K. S. Trivedi. Modeling and Quantification of Security Attributes of Software Systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pages 505–514, June 2002.

[MMS85] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic Activity Networks: Structure, Behavior, and Application. In *Proceedings of the International Workshop on Timed Petri Nets*, pages 106–115, July 1985.

[ODK99] R. Ortalo, Y. Deswarte, and M. Kaâniche. Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security. *IEEE Transactions on Software Engineering*, 25(5):633–650, 1999.

[RPL+02] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pages 229–238, June 2002.

[SM01] W. H. Sanders and J. F. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. In E. Brinksma, H. Hermanns, and J. P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, pages 315–343. Springer-Verlag, Berlin, 2001.

[TCS85] US Department of Defense 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.