Probabilistic Validation of an Intrusion-Tolerant Replication System*

Sankalp Singh†, Michel Cukier‡, and William H. Sanders†

University of Illinois†
Urbana, IL 61801, USA
{sankalps,whs}@crhc.uiuc.edu

University of Maryland‡
College Park, MD 20742, USA
mcukier@eng.umd.edu

Abstract

As computer systems become more complex and more widely distributed, it is becoming increasingly difficult to remove all vulnerabilities that can potentially be exploited by intruders. Intrusion tolerance is an emerging approach that aims to enable systems to continue functioning in spite of successful intrusions. Before intrusion tolerance is accepted as an approach to security, there must be quantitative techniques to measure its efficacy. However, there have been very few attempts at quantitative validation of intrusion-tolerant systems or, for that matter, of security in general. In this paper, we show that probabilistic validation through stochastic modeling is an attractive mechanism for evaluating intrusion tolerance. We demonstrate our approach by using stochastic activity networks to quantitatively validate an intrusion-tolerant replication management system. We characterize the intrusion tolerance provided by the system through several measures defined on the model, and study variations in these measures in response to changes in system parameters to evaluate the relative merits of various design choices.

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. Intrusion tolerance 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.

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. Most traditional approaches to security validation have not been quantitative (e.g., the Security Evaluation Criteria [14]). Quantitative methods, when attempted, have either been based on formal methods [7], 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., [9]) to try to compromise a system. Both approaches, while being valuable in identifying system vulnerabilities, have their limitations.

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. Probabilistic evaluation has been used extensively in the dependability community, but very few attempts have been made to use it to assess system security. Early work on probabilistic quantification of security was done by Littlewood et al. [8]. That exploratory work primarily suggested questions that must be answered in order to make probabilistic security evaluation viable. Jonsson et al. [6] conducted several experiments and presented a quantitative model of a security intrusion based on attacker behavior. Their approach considers only one source of uncertainty in security validation: the behavior of the attacker. Several attempts have been made to build models that take into account the attacker as well as the system being validated. For example, Gong et al. [4] present a general 9-state model of an intrusion-tolerant system for describing known and unknown attacks. Jha and Wing [5] use a state machine model with injected faults and a survivability property specified using temporal logic to generate a scenario graph, which is then used for evaluating overall system reliability or latency using Bayesian networks. Or-

*This research has been supported by DARPA contract F30602-00-C-0172.

talo et al. [10] propose modeling known vulnerabilities in a system combined with simple assumptions concerning an attacker's behavior, which can be analyzed using standard Markov techniques once several parameter values have been obtained experimentally.

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. 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 probabilistic models for intrusion-tolerant systems should, either explicitly or logically, include submodels of the attacker, the intrusion-tolerance mechanism being used, the application, and the resource/privilege state of the system. It is also important to determine the appropriate level of detail/abstraction. 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. Furthermore, depending on the nature of the attack, the attacker model may either represent details of the intrusion itself (corresponding to explicit representation of faults in a dependability model) or represent the *effect* of the intrusion (corresponding to the representation of errors in a dependability 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 paper, we show that probabilistic modeling using Stochastic Activity Networks (SANs) [12] 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.

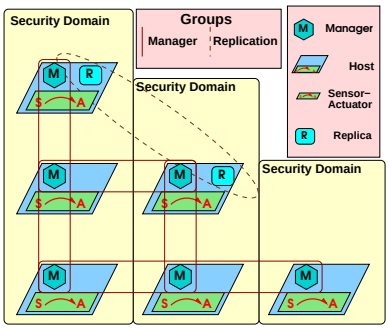


Figure 1. ITUA Architecture

The remainder of the paper is organized as follows. First, Section 2 provides a brief overview of the ITUA replication system and the assumptions that were made in constructing the model. Section 3 describes the composed stochastic activity network representation of the model for the system described in Section 2. Section 4 gives the various results we obtained from the model, along with our interpretations and inferences. We conclude in Section 5 with a synopsis of the major contributions of the paper.

2. Overview of ITUA Replication System and Model Assumptions

The Intrusion Tolerance by Unpredictable Adaptation (ITUA) [2] 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 also 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 1. Each domain implements a boundary that the attackers have difficulty crossing.

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 system is used to multicast among replica groups and the manager group [11].

The ITUA replication management system has three major functions: 1) making decisions about the group structure of the managers and replicas; 2) propagating information regarding important state changes among the managers, so that the managers are aware of the state of the system in order to make decisions; and 3) convicting corrupt members of the system to prevent 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 intrusions, 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 [1] 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 also result in fewer Byzantine faults being tolerated.

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 it 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, provided that less than a third of the currently active group members are corrupted. The replication group excludes the convicted replica from all future communications, and each correct replica in the group 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 the currently active managers are corrupt), they randomly pick the domain in which to start the new replica. 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, 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, by excluding all the hosts in the domain, including their replicas and management entities. 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 by using the corrupt hosts for covert purposes. We have also considered an alternative approach in which only the host running the corrupt replica is excluded, not the entire domain. We assume the system is left to itself with minimum human intervention; hence, we do not model manual repair of excluded domains/hosts, and can run out of domains/hosts to start new replicas to replace the killed ones.

We make several assumptions about attacker behavior. We have based our attacker model on the experiments conducted by Jonsson et al. [6], 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.

We assume that a secure mechanism for starting replicas on a chosen host is in place. To keep the model simple, we also assume that the middleware starts the same number of replicas for each application, and also that each security domain has the same number of hosts.

3. SAN Models

We use Stochastic Activity Networks (SANs) [12] as our modeling formalism. We built atomic SAN submodels for a single host, a single replica, and a management algorithm. A complete model of the system was then composed using replicate and join operations. We now give a description of the salient features of the model. A more detailed treatment can be found in [13].

3.1. Composed Model

Figure 2(a) 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 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). Similarly, the **Host** submodel is replicated `num_hosts` times to form a *security domain* submodel, which is then replicated `num_domains` times (*Rep2* in the figure) to model `num_domains` security domains. 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.

3.2. SAN Model for an Application Replica

Figure 2(b) 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

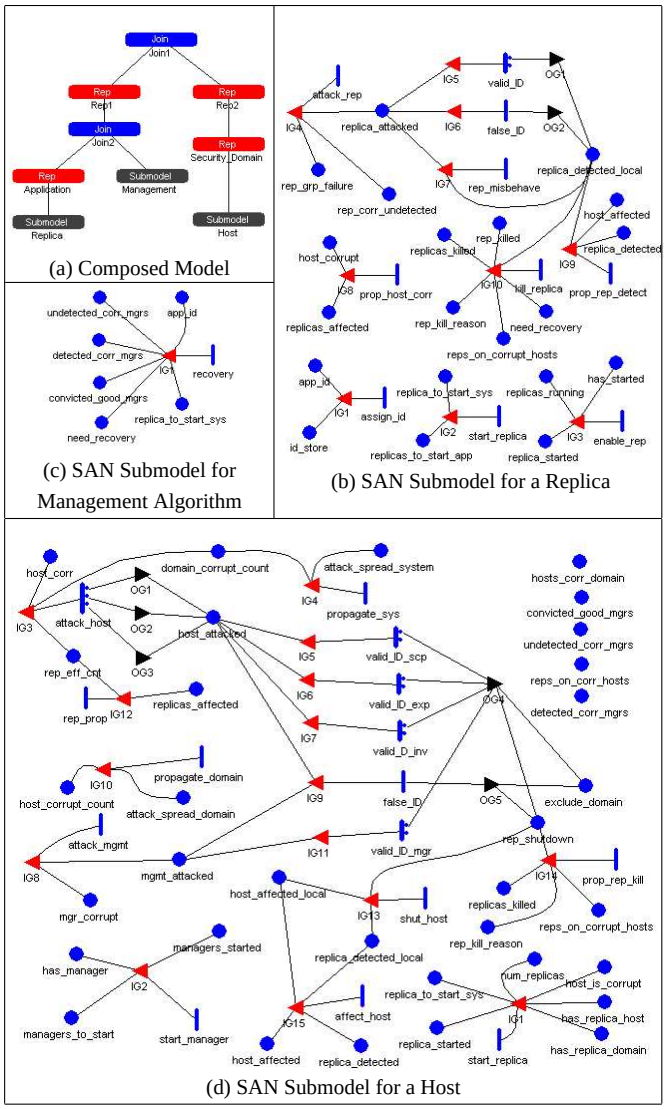


Figure 2. SAN Models

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.

We correlate between replicas and the hosts on which they are running using various shared places that act as bit vectors; we discern among the replicas of different applications by associating a unique identifier (a particular bit position) with each application. The high-rate activity *assign_id* fires repeatedly as soon as the model is solved or simulated, until each application has received a unique identifier of the form 2^n, $0 \leq n \leq 14$ stored in the place *app_id*, which is shared by all replicas of an application. Since places hold `short` integers, that assignment of identifiers limits the number of applications, to at most 15, but we believe that is more than enough for most studies.

The next step in the initialization of the model is to start replicas on hosts. The place *replicas_to_start_sys* is a bit

vector shared across all *Replica* and *Host* submodels, and indicates the applications for which replicas are to be started by the *Host* submodels. The high-rate activity *start_replica* fires once for each *Replica* submodel, setting the bit corresponding to the identifier of the application to which the replica belongs (*app_id*) in *replica_to_start_sys*. 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* activities are enabled in all *Replica* submodels that correspond to the application's replicas that have not yet started, with each activity being equally likely to fire first. The first *enable_rep* 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 this replica and indicates if the replica represented by this submodel is active, to one; 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 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* (since a host, and for some management schemes an entire domain, can have at most one replica of a particular application). For each affected application, the activity *prop_host_corr* fires in one of its *Replica* submodels that is not already running on a corrupt host, changing the state of the *Replica* to indicate that its host is corrupt (local place *host_corrupt*), and resetting the bit for this application in *replicas_affected*.

The activity *attack_rep* represents a successful attack on a replica. The rate of the activity (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 (*host_corrupt*). We multiply the base rate by a constant to obtain the higher rate. A multiplier of 2 would imply 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 a third or more of the total number of application replicas currently running (*replicas_running*), the marking of *rep_grp_failure*, which is shared across all replicas of the application and is used to determine the "unreliability" of the application, is set to 1.

After an intrusion into a replica, the activity *valid_ID* is enabled. The activity has two cases, which represent successful detection and failure to detect, respectively. Upon successful detection the marking of *rep_corr_undetected* is decremented and the marking of *replica_detected_local* is set to 1. A corrupt replica may exhibit anomalous behavior, which can be detected by other currently running replicas of the application provided that enough of them are correct. This is captured by activity *rep_misbehave*. 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 activity *false_ID* models the false alarms of replica corruption generated by the intrusion detection system. It is enabled whenever the replica has been intruded. The results of the firing of *false_ID* and *rep_misbehave* are similar to those of the firing of *valid_ID*.

Once a replica is marked as corrupted (via valid intrusion detection, false alarm, or detection of misbehavior by the replication group), the activity *prop_rep_detect* conveys that information to the host on which the replica is running, setting the replica's application identifier bit in the globally shared place *rep_affected*, and copying *host_corrupt* into the globally shared *host_affected*, to indicate the state of the host on which the replica is running. 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 indicating the applications whose replicas were killed; the marking of *rep_kill_reason* is a bit vector indicating the applications that had compromised replicas on the host (or domain); and the marking of *reps_on_corrupt_hosts* is a bit vector indicating the applications whose replicas were running on corrupt hosts in the domain being shut down. The markings of those places are used to determine the appropriate replicas to kill. The activity *kill_replica* fires in those *Replica* submodels, decrementing the number of active replicas (*replicas_running*), resetting the markings of various local places in the *Replica* submodel so that the submodel can be used again to start a new replica of the application, and incrementing the marking of *need_recovery* to indicate that the management infrastructure must start a new replica for this application.

3.3. SAN Model for Management Algorithm

Figure 2(c) shows the SAN representation of the **Management** submodel. This SAN models the process of re-

covery by the management infrastructure through the starting of new replicas to replace those killed due to domain and host exclusions. As shown in Figure 2(a), there is one *Management* SAN per application. The activity *recovery* is enabled whenever there are some replicas to be started for the application (indicated by the marking of *need_recovery*, shared with all *Replica* submodels for the application), and there are enough good managers in the system to initiate a recovery (i.e., if the number of undetected corrupt managers is less than a third of the total number of managers currently running). 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.

3.4. SAN Model for a Host

Figure 2(d) shows the SAN representation of the **Host** submodel. This SAN models the activities on a single host, including 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, to name a few.

The high-rate activity *start_manager* is responsible for starting a manager on each host. The high-rate activity *start_replica* is responsible for starting replicas on hosts. The activity is enabled whenever there is an application for which there is a replica to start (indicated by the globally shared bit vector *replica_to_start_sys*) and for which there is not already a replica in the domain, and the domain has not been excluded yet. Since the identical copies of the activity would be enabled in all *Host* submodels for which those conditions are met, all of the copies are equally likely to fire first. Hence, to start a replica, we choose a domain uniformly from among the domains that qualify, and within the chosen domain, we select a host uniformly from among the hosts 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 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 conveyed to the appropriate *Replica* submodels via the globally shared bit vector *replica_started*. The corruption state of the host is also conveyed to those replicas via the globally shared place *host_is_corrupt*.

As mentioned in Section 2, the attacker can attack the host (i.e., the host operating system and services), the management infrastructure, and the application replicas. The activity *attack_host* models attacks on the host operating system and services. As mentioned in Section 2, an attack on the host can belong to one of three categories, script-based, more exploratory, and innovative, which are modeled by three *cases* for the activity *attack_host*, configured with decreasing probability. Upon the firing of *attack_host*, the marking of *host_attacked* is 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. The activity *rep_prop* is enabled when the host is intruded, and upon firing of the activity, the marking of globally shared place *replicas_affected* is set to a bit vector indicating the applications that have replicas on this host. It is then used by the *Replica* SANs as already described.

As mentioned in Section 2, 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*, which models the spread of an attack within a domain, and *propagate_sys*, which models the spread of an attack across domain boundaries. Both activities fire exactly once when a host becomes intruded. 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 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* increases linearly with the markings of *attack_spread_domain* and *attack_spread_system*, increasing the chances of successful intrusions into the host operating system and services. The spread effect variable associated with intra-domain spread is set to be much larger than the variable associated with inter-domain spread.

The activity *attack_mgmt* represents attacks against the manager running on the host. The rate of this activity increases if the host is corrupted. 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 set to be higher for script-based attacks than for more exploratory

attacks, which in turn is set to be higher than the value 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, a response is initiated provided that the manager on the host and the manager group of the domain are not corrupt. The domain containing the corrupt host is excluded by placing a token in *exclude_domain* that is shared across the domain.

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.

As mentioned in Section 3.2, 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*, and *host_affected* is changed to indicate if the host on which the detected replica was running was corrupt. The activity *affect_host* can fire if the host has a replica of the application indicated by *replica_detected* and the host's corruption status matches with that conveyed by *host_affected*, and the firing moves values of *replica_detected* and *host_affected* into domain-level shared places *replica_detected_local* and *host_affected_local* to avoid the possibility of a deadlock in the model. The activity *shut_host* is enabled if there is some convicted replica (*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., less than a third are in the undetected corrupt state), then they will exclude the corrupt manager and its domain. The actual shutting down of the domain is modeled by the activity *prop_rep_kill*, which sets the markings of *replicas_killed*, *rep_kill_reason*, and *reps_on_corrupt_hosts* as mentioned earlier in Section 3.2.

We have also modeled an alternative management algorithm that excludes from the system 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. We briefly summarize the salient differences. 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, that information was sent about the domain.)

4. Results

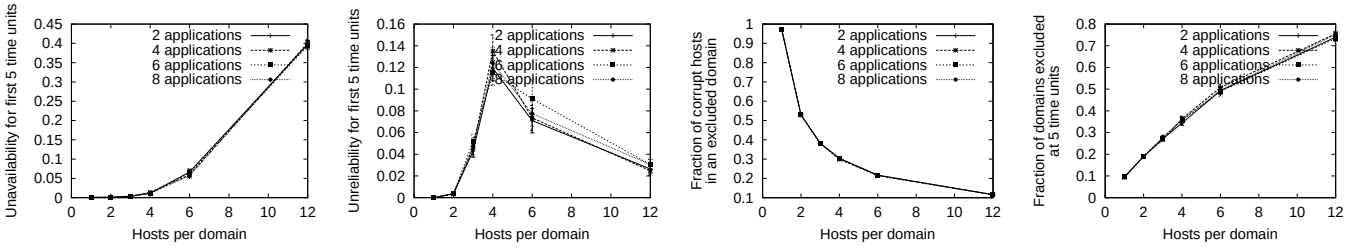
We used the Möbius [3] tool to design the SANs, define the intrusion tolerance measures on the model, and design studies on the model. Möbius can solve SANs analytically by converting them into equivalent continuous time Markov chains. However, because of the complexity of the model and the use of non-exponentially 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. We defined the service by an application to be improper if it suffers a Byzantine fault, i.e., a third or more of the currently active replicas are corrupt. Some of the measures defined were *unavailability* for an interval, which is the fraction of time the service was improper in the interval; *unreliability* for an interval, which is the probability that service was improper at least once in the interval; *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 when it is excluded; and *fraction of excluded domains* at a given time instant.

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. A more detailed analysis, along with a larger set of studies, can be found in [13].

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.



(a) Unavailability (b) Unreliability (c) Fraction of Corrupt Hosts in an Excluded Domain (d) Fraction of Domains Excluded

Figure 3. Variations in Measures for Different Distributions of 12 Hosts for First 5 Hours

- 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 security literature. In the following studies, we have also tried to explore the system's sensitivity to variations in these parameters.

4.1. Impact of 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 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.

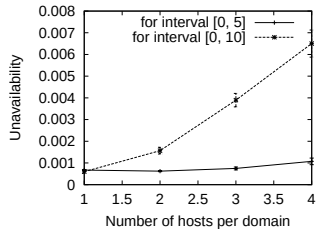
Figure 3(a) shows the variation of unavailability of an application for different distributions of the 12 hosts. An important point to note is that as we move along the X-axis, the number of domains reduces as the number of hosts per domain increases. As is evident from the graphs for unavailability 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.

Figure 3(b) 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 a 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.

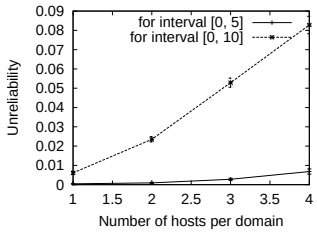
Figure 3(c) shows the fraction of hosts that were infiltrated in a domain by the time it was excluded. Resources are wasted when we have more hosts per domain, since corruption of a very small fraction of hosts results in exclusion of a large number of uncorrupted hosts. Note that the fraction is not 1 when we have one host per domain, since false alarms can result in some domains being excluded without any host being corrupted. Figure 3(d) shows that a large number of domains were excluded at the end of 5 hours when we had more hosts per domain, adversely affecting the availability of the applications.

4.2. Impact of Different Numbers of Hosts Distributed into a Constant Number of Domains

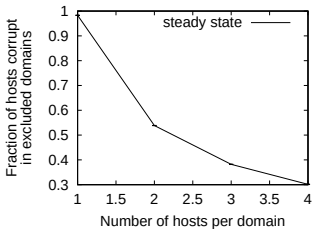
The above studies indicate that for a given set of hosts, it is best to distribute them into as many domains as possible. To determine the gains, if any, to be obtained by putting more hosts per domain into a fixed number of domains, and to judge the cost/benefit tradeoff, we conducted a second study 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,



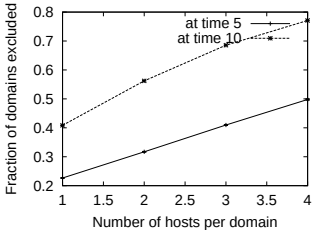
(a) Unavailability



(b) Unreliability



(c) Fraction of Corrupt Hosts in an Excluded Domain



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

Figure 4. Variations in Measures for Different Numbers of Hosts in 10 Domains

in the second study, the total number of hosts changed for each experiment in the study.

Figures 4(a) and 4(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 5 time units is quite low, and even for the first 10 time units, it is much lower than the variation observed in the study described in Section 4.1.

Figure 4(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 4(d) indicates that the number of domains that have been excluded increases due to the increase in the number of hosts. That 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.

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

4.3. Comparison of Domain-exclusion and Host-exclusion Management Algorithms

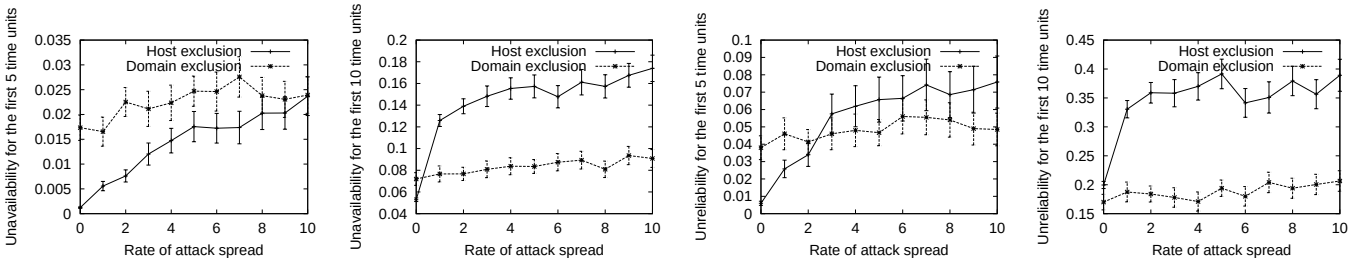
Another major management issue is that of what to exclude when a host (or a replica on it) is found to be corrupt (assuming multiple hosts per domain). One approach is to exclude the entire domain that contains the host. This is a preemptive strike against the attackers, using the assumption that the 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. 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(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. On the other hand, as shown in Figure 5(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.

Figure 5(c) shows that under the domain-exclusion scheme, application reliability doesn't 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(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 low within-domain attack spread rate, a preemptive-action-based domain-exclusion scheme performs almost as well as the host-exclusion scheme in the



(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. Unavailability and Unreliability for Different Exclusion Algorithms

short run, and significantly better in the long run.

5. Conclusion

In this paper, we present 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, than it is to analyze the same properties of the real system. The SAN model created was modular, and it can be easily modified to represent other intrusion-tolerant systems.

Furthermore, 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 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.

Acknowledgments: We would like to thank the other members of the ITUA team for their helpful comments. We are grateful to Jenny Applequist for her editorial assistance.

References

[1] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. Third Symp. on Operating Systems Design and Implementation*, pages 173–186, Feb. 1999.

[2] 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 Intl Conf. on Dependable Sys. and Networks (DSN-2002)*, pages C–5–1– C–5–3, June 2002.

[3] 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 Trans. on Software Engineering*, 28(10):956–969, Oct. 2002.

[4] F. Gong, K. Goseva-Popstojanova, F. Wang, R. Wang, K. Vaidyanathan, K. Trivedi, and B. Muthusamy. Characterizing Intrusion Tolerant Systems Using A State Transition Model. In *Proc. DARPA Information Survivability Conf. and Expo. II (DISCEX'01)*, pages 211–221, 2001.

[5] S. Jha and J. M. Wing. Survivability Analysis of Networked Systems. In *Proc. 23rd Intl Conf. on Software Engineering (ICSE2000)*, pages 307–317, 2001.

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

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

[8] 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.

[9] J. Lowry. An Initial Foray into Understanding Adversary Planning and Courses of Action. In *Proc. DARPA Information Survivability Conf. and Expo. II (DISCEX'01)*, pages 123–133, 2001.

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

[11] 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 *Proc. 2002 Intl Conf. on Dependable Systems and Networks (DSN 2002)*, pages 229–238, June 2002.

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

[13] S. Singh. Probabilistic Validation of an Intrusion-Tolerant Replication System. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

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