An architecture for adaptive intrusion-tolerant applications

Partha Pal[1,*] and Paul Rubel[1], Michael Atighetchi[1], Franklin Webber[1], William H. Sanders[2], Mouna Seri[2], HariGovind Ramasamy[3], James Lyons[2], Tod Courtney[3], Adnan Agbaria[2], Michel Cukier[3], Jeanna Gossett[4], Idit Keidar[5]

[1] *BBN Technologies, Cambridge, Massachusetts. {ppal, prubel, matighet, fwebber}@bbn.com*
[2] *University of Illinois at Urbana-Champaign. {whs, seri, ramasamy, jlyons, tod, adnan}@crhc.uiuc.edu*
[3] *University of Maryland at College Park, Maryland. mcukier@eng.umd.edu* [4] *The Boeing Company. jeanna.m.gossett@MW.Boeing.com* [5] *Department of Electrical Engineering, Technion – Israel Institute of Technology. idish@technion.ac.il*

SUMMARY

Applications that are part of a mission-critical information system need to maintain a usable level of key services through ongoing cyber-attacks. In addition to the well-publicized denial of service (DoS) attacks, these networked and distributed applications are increasingly threatened by sophisticated attacks that attempt to corrupt system components and violate service integrity. While various approaches have been explored to deal with the DoS attacks, corruption-inducing attacks remain largely unaddressed. We have developed a collection of mechanisms based on redundancy, Byzantine fault tolerance, and adaptive middleware that help distributed, object-based applications tolerate corruption-inducing attacks. In this paper, we present the ITUA architecture which integrates these mechanisms in a framework for auto-adaptive intrusion-tolerant systems, and describe our experience in using the technology to defend a critical application that is part of a larger avionics system as an example. We also motivate the adaptive responses that are key to intrusion tolerance, and explain using the ITUA architecture how to support them in an architectural framework.

KEY WORDS: Intrusion Tolerance, Byzantine Fault Tolerance, Adaptive Defense, Redundancy, Adaptive Middleware, Survivability Architecture

*Correspondence to: BBN Technologies, 10 Moulton Street,Cambridge, MA 02138
Contract/grant sponsor: DARPA; contract/grant number: F30602-00-C-0172

Introduction

An Intrusion-Tolerant System (ITS) aims to maintain a useful level of operational capability throughout ongoing cyber-attacks. The applications that are part of an ITS, especially those that provide critical services for the system's mission therefore, must survive the (partial[†]) failures and unwanted changes in the system caused by malicious acts of intruders.

Most attacks result in undesirable and harmful consumption, corruption, or control of resources such as bandwidth, memory, CPU cycles, processes, or hosts. These either lead directly to the attacker's goal (e.g., deny a service, or breach the integrity of the service), or leave them in a better position by increasing their privileges in the system. Therefore, having redundancy in the system, by itself, is not sufficient in an ITS. It must cope with these changes either by actively engaging supplementary mechanisms to counter or remedy the effects, or by continuing despite the changed situation (perhaps providing a degraded level of service). *Adaptation* is used in this paper to mean system activities that are not directly part of the system's principal functional behavior, but are crucial to keeping the system operational through attacks. It is our thesis that strategic use of adaptive response, integrated into the application and supported by redundancy, makes a significant level of intrusion tolerance achievable.

How much redundancy is required and where, as well as which adaptive response needs to be mounted and when, are determined by an application-specific "survivability strategy," which is based on the system's survivability requirements. Some responses are autonomic, while others require human intervention. Some responses will be local in effect and require little coordination, while others will have a more global effect and require system-wide coordination.

As observed in [16, 17, 39], attackers often attempt to manipulate the defense, rather than confront it directly, to achieve their objectives. Key defensive elements such as the survivability strategy and its implementation that controls the overall adaptation are therefore, a prime target of exploitation for the attacks. To harden that target, the adaptation control mechanism should be a distributed computation, implemented by redundant management elements that are somewhat suspicious of each other. Distribution implies that attacks must strike at multiple places to have an impact; redundancy implies that sacrifice of a small number of elements will not make the system ineffective as a whole; and mutual suspicion, implemented in the form of signatures and consensus, implies that it is harder to fool or take over the ITS by corrupting a few management entities.

The research discussed in this paper was performed under the ITUA[‡] project, where we developed a collection of mechanisms based on Byzantine fault tolerance, redundancy, and adaptive response that facilitates tolerance of sophisticated attacks by intruders[§] attempting to spread corruption in the system. The ITUA architecture is a distributed objects framework

[†]Much like fault tolerance, if the intrusion-induced failure is total and pervasive in the system, then no autonomic intrusion tolerance is possible and manual recovery is required.
[‡]ITUA stands for Intrusion Tolerance by Unpredictability and Adaptation.
[§]Insider and physical attacks are not addressed by ITUA.

for integrating these mechanisms in the defense of an individual application[¶]. The mechanisms provide a range of adaptive responses, available across multiple system layers, as well as at redundant and distributed management objects. By incorporating adaptive capabilities within the defended (often called defense-enabled) application, the architecture facilitates isolation of compromised resources, failure recovery, and graceful degradation. Cryptographic techniques are used for stronger authentication, and for signing messages in consensus algorithms. Consensus helps minimize the risk of being compromised by a single trusted entity. Intrusion-tolerant gateways are used to protect communication between objects. To make it more difficult to exploit the defense, ITUA injects uncertainty, from the attacker's point of view, into its adaptive responses.

We also developed a methodology for validating intrusion tolerance quantitatively, and applied it to validate the ITUA architecture. Our methodology uses complementary methods: 1) testing a prototype implementation, 2) allowing security analysts other than the implementers to search for flaws in the architecture and in the prototype, 3) model checking of key parts of the distributed algorithms, and 4) modeling the architecture and estimating its tolerance to attack by studying the model's properties. Space does not permit us to discuss survivability validation methods and results here, but initial results reinforce our thesis and provide positive indication that the architectural approach is useful. Details on each of the methods and their results can be found in [8, 10, 30].

This paper presents the ITUA architecture, the individual mechanisms that are part of the architecture and the adaptive capabilities that they collectively enable, along with our experience in using the architecture to defend a critical application that is part of a larger avionics system. The rest of the paper is organized as follows. Section 2 describes related work. Section 3 presents a motivating example, followed by an overview of the ITUA architecture in Section 4, explaining the need for supporting the key adaptive capabilities in the architecture. Section 5 describes the details necessary to understand and implement an architectural framework for intrusion tolerance using adaptation and redundancy. Section 6 is a discussion of our experience in applying the ITUA architecture in the context of a Boeing avionic example, and lessons learned. Section 7 concludes the paper.

Related Work

We first showed the feasibility of integrating defensive adaptation into an application in 1999 [13]. Since then, many researchers, including ourselves [28, 26, 27, 39], continued to explore various ways of using adaptive techniques in cyber-defense, and auto-adaptive capabilities have now become almost a standard feature in intrusion-tolerant systems. Below is a sample of intrusion tolerance research that utilize autonomic adaptation in a manner similar to ours.

Intrusion tolerance in the **Willow** architecture [12] is achieved using a combination of 1) disabling of vulnerable network elements when a threat is detected or predicted, 2) replacement

[¶]This enables the defended application to become part of a larger ITS.

of failed system elements, and 3) reconfiguration of the system if non-maskable damage occurs. Willow uses Control Loops to perform proactive and reactive control actions that are similar to the adaptive responses in ITUA. Willow appears to be more scalable than ITUA in its use of the publish-subscribe paradigm for coordination, but does not address corruption (i.e., attacks on integrity). It has also taken a more centralized position by having a dedicated analysis/diagnosis component and reconfiguration scheduler in contrast to ITUA's distributed management of adaptation and defense strategy.

ITDOS [33] integrates a Byzantine fault-tolerant protocol into an open-source CORBA ORB to provide an intrusion-tolerant middleware. This foundation allows up to f simultaneous Byzantine failures in a system of at least $3f + 1$ replicas. Voting is performed on unmarshalled CORBA messages, allowing heterogeneous application implementations for a given service and thereby increasing implementation diversity. Symmetric session keys provide confidential client-server communications. The ITDOS Byzantine protocol is tightly integrated with a specific ORB via pluggable protocols, whereas the ITUA Byzantine protocols operate above the ORB in a gateway. Moreover, ITUA has more sophisticated replica management algorithms, and shows defense in more system layers (defense in depth).

DIT [38] comprises functionally redundant COTS servers running on diverse operating systems and platforms, hardened intrusion-tolerant proxies that mediate client requests and verify the behavior of server and other proxies, and monitoring and alert management components based on the EMERALD [20] intrusion-detection system. The system adapts its configuration dynamically in response to intrusions or other faults. DIT does not employ Byzantine fault-tolerant replication protocols, but has a range of adaptive responses to isolate compromised parts of the system.

MAFTIA [1] is a European project developing an open architecture for transactional operations on the Internet. MAFTIA models a successful attack on a security domain, leading to corruption of processes in that domain, as a "fault"; the architecture then exploits approaches to fault tolerance that apply regardless of whether the faults have an accidental or malicious cause. MAFTIA is explicitly middleware-based and provides both protection from and tolerance of intrusions. Like ITUA, MAFTIA makes use of partial synchrony assumptions and hybrid fault models. While both MAFTIA and ITUA took an architectural approach to intrusion tolerance and consider malicious attacks that can cause Byzantine behavior in the system, their target applications differ: MAFTIA focuses on transactional operations on the Internet, while ITUA focuses on distributed object-based applications that run in a critical infrastructure network. Therefore, ITUA architecture has elements (such as the gateway) and protocols (such as kill or start replica) designed for distributed object environments (i.e., ORBs) as opposed to MAFTIA's transactional and application service oriented environment. ITUA also makes assumptions about the network characteristics (predictable latency variations, and availability of bandwidth) that are not suitable for an Internet-like environment.

The IEIST Application - A Motivating Example

IEIST (Insertion of Embedded Infosphere Support Technologies) [4], an advanced avionics system (see Figure 1) developed at Boeing focuses on the development of off-board software agents, called Guardian Agents (GAs), that augment embedded tactical systems and plug into the evolving Joint Battlespace Infosphere (JBI) [37], while still providing interoperability with legacy systems and communication links. A typical IEIST application involves the interactions among a fighter Guardian Agent, an Unmanned Combat Air Vehicle (UCAV) Guardian Agent, and a Discovery and Navigation Fuselet (henceforth called Fuselet[‖]). These three components are highlighted (by surrounding rings) in Figure 1 to distinguish them from other IEIST components. The UCAV registers with the Fuselet as a publisher for the region it monitors. The fighter registers with the Fuselet as a subscriber for the region in its flight path. These interactions are denoted by the dotted connectors marked 1. The Fuselet determines if the fighter has a matching publisher, and if it does, connects the appropriate UCAV to the fighter. Once this connection is established, activities noted by the UCAV are sent directly to the fighter. This is denoted in the figure by the dotted connector marked 2. Multiple fighters and UCAVs are served by the same Fuselet. The Fuselet is a critical component of the application because it enables the fighters to hook up with the UCAVs that can provide important information about their target and flight path. Note that mounting a Denial of Service (DoS) on the Fuselet is not likely to be attacker's first preference since this will be a noisy attack, making the UCAVs and fighters aware of the problem and allowing them to make conservative decisions. A much more fatal attack is when the Fuselet is corrupt and provides wrong information to the guardians.

An intruder interested in damaging the application this way is expected to start with reconnaissance activities trying to identify IP addresses and ports that can be exploited to obtain a foothold in the system. A properly defended system minimizes such exposure, but some IP addresses and ports must be open for the application's own interaction and interoperability. These could be defended by adaptive blocking of IP addresses and ports. The attacker who has infiltrated a host running the Fuselet may try to introduce corruption by manipulating data and files on the local disk, which could be countered by monitoring the critical data and files, and restoring the damaged ones. But a general approach to defending a critical application component like the Fuselet against such corruption is to make it Byzantine fault-tolerant. This involves replication and replica coordination protocols. Nevertheless, a determined attacker will be successful in corrupting some replicas, and the defended system must strive to minimize rapid propagation of corruption among replicas by using adaptation (for instance, quickly removing a corrupt replica from the system) and diversity (for instance, different OS platforms in the architecture will prevent attacks that exploit the vulnerabilities of a particular OS from being successful at all replica locations). Finally, the defended system should be prepared for the possibility that a significant number of replicas and hosts

[‖]A Fuselet is a specialized entity in the Joint Battlespace Infosphere, whose primary responsibility is to transform information gained from observed data into useful knowledge, for instance, deducing that UCAV x needs to talk to fighter y from the published fighter routes and UCAV monitoring regions.

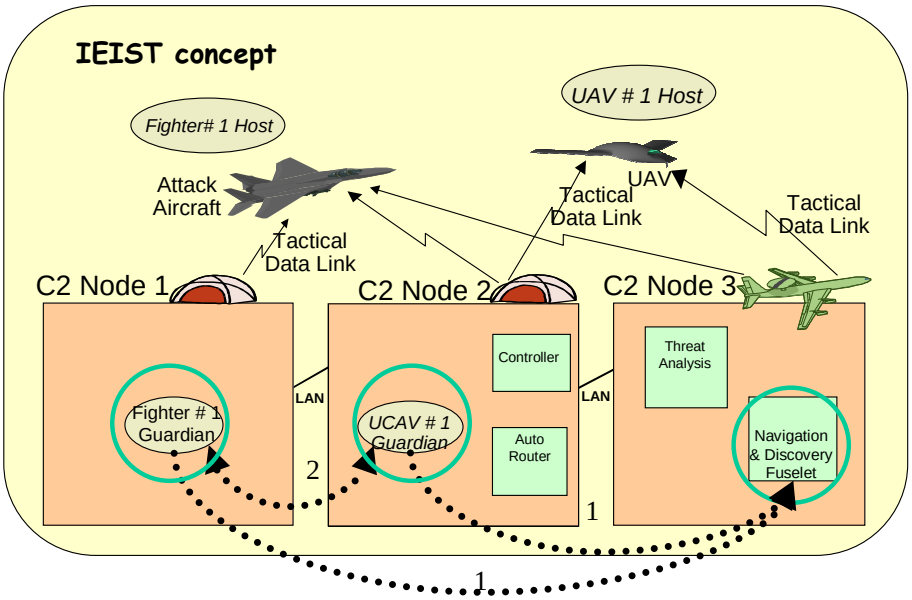

Figure 1. A Typical IEIST Application

may eventually be compromised by the attack making continued Byzantine fault tolerance impossible.

Overview of the ITUA Architecture

The ITUA architecture is a framework in which many of the above-mentioned issues are addressed by a coherent integration of a number of individual defense** mechanisms. This section introduces the key architectural elements, and motivates the design decisions we made in creating the architecture. More details about the architecture and its capabilities can be found later in this paper.

Figure 2 is a notional view of the ITUA architecture, describing its key elements in terms of the intrusion-tolerant version of the IEIST application introduced earlier††. It shows redundant hosts organized into security domains, and Fuselet replicas (only two replicas are shown, at

**We use the term *defense mechanism* rather broadly: any mechanism that is used in cyber-defense is a defense mechanism in our terminology. These mechanisms could be further categorized as prevention, avoidance, tolerance or other mechanisms.

††A generic view of the ITUA architecture in terms of a client-server application can be obtained by simply thinking of the Fuselet as the server and the guardians as clients.

least four are used in practice) organized in a group. The guardians are not replicated, and only one guardian is shown. A redundant number of management objects, known as the ITUA managers (henceforth called "managers", and annotated as "Mgr" in Figure 2), collectively perform coordination of system-wide adaptation and management of redundant resources.

To ensure that compromise (of availability and/or integrity) at a small number of key application (e.g., the Fuselet replicas) and management (e.g., the managers) objects are not fatal for the application's continued operation, a redundant number of these objects are used with Byzantine fault-tolerant algorithms for inter-object coordination. In order to contain the attack effects, redundancy is managed adaptively so that compromised replicas are removed and corrupt hosts are isolated. The current implementation of the architecture does not support replenishment of lost hosts[‡‡] as an adaptive response, but autonomic replenishment of lost replicas are supported. To further reinforce the defense, the resistance and tolerance mechanisms are implemented at multiple system layers as indicated by the internal details of a manager or a replica in Figure 2. This combination of redundancy and layering imply that the attacker has to corrupt multiple system layers at multiple hosts in order to achieve his objectives.

However, replication of the Fuselet and replenishment of compromised Fuselet replicas alone are not unsurmountable. If there is no diversity in the system, the attacker may corrupt multiple replicas at the same time or keep corrupting newly introduced replicas using the same attack steps. While acknowledging the potential benefits of physical diversity such as different OS platform or diverse Fuselet implementations, ITUA explores other measures to introduce artificial diversity. It appears to be more practical (from the developer's (Boeing) and user's (Air Force) point of view of a system like IEIST) to enforce the notion of security domains, which create separation among redundant hosts by means of security administration, than to port a propriety application like the Fuselet on multiple OSs, or to develop, maintain and administer multiple implementations of the Fuselet on multiple platforms. While the diversity introduced by the security domains makes it harder for the attacker to cross domain boundaries, another form of diversity introduced by injecting uncertainty in adaptive response (such as unpredictable placement of new replicas), creates additional hurdles for the attacker who wants to attack the next new replica. The ITUA architecture is amenable to physical diversity, and some of the key infrastructural components of the architecture, such as the managers (written in Java) and the gateways run on multiple OS platforms.

As a final complement to the multi-layered and adaptive defense, the ITUA architecture enables the application components such as the guardians to change their behavior gracefully when loss of a significant number of Fuselet replicas and hosts makes Byzantine fault tolerance untenable. In such a situation, the guardians adapt to interact with stand-alone instances of the Fuselet that are maintained in the architecture as a backup.

Key elements of the ITUA architecture are formally introduced below.

[‡‡]A scheme is under investigation where an isolated host can be re-instated after a clean reboot. In the mean time, new hosts can be added to the architecture manually.

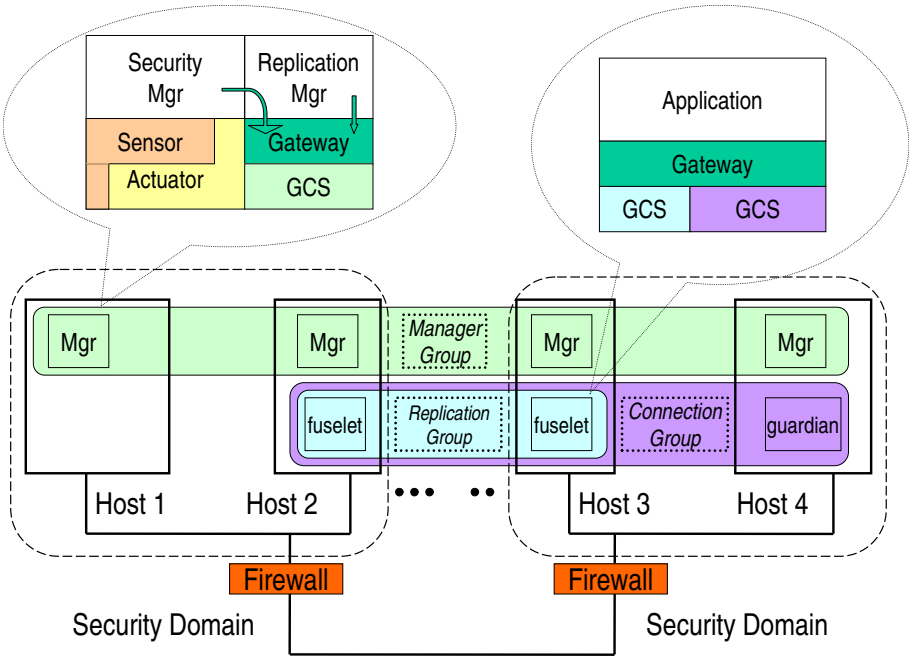

Figure 2. Notional ITUA Architecture

Security Domains

Security domains, implementing boundaries that attackers have difficulty crossing, are the basis for organizing redundant hosts in the ITUA architecture. A host can be in a security domain by itself, or a domain may consist of a set of hosts. If an attacker has privileges on a host h in a security domain D, he is not automatically granted privileges on hosts in other security domains. This makes it much harder for a compromised host within a security domain to spread the attack to other domains than to compromise other hosts within its own domain. The number of security domains, the number of hosts within a domain, and the mechanism used to enforce the the isolation between domains are determined based on the application's needs, and form an important part of its survivability strategy. Enforcement mechanisms can range from simple administrative techniques ensuring that hosts have different passwords to more complicated management of firewall policies. Our validation studies [10, 36] indicate that it is better for survivability to have a smaller number of hosts in a security domain.

Application Objects

As shown in Figure 2, application objects (such as Fuselet and guardians) are one of the two (managers comprise the other kind) kinds of top level run time elements in the ITUA architecture. Both replicated and non-replicated application objects must be supported in the architecture. In general, service consumers like the guardians (which is most likely operated by a human operator), are not as critical as the service provider objects like the Fuselets, whose availability and integrity are essential for the application. Therefore, in instances like the IEIST application, replicating the service provider (such as the Fuselet) object(s) is sufficient. That is why Figure 2 shows Fuselets replicas, but an unreplicated guardian. However, this may not be true in a more object-to-object situation (i.e., each is issuing and respondng to service requests) where both peers may need to be replicated.

One way to manage replicas of application objects is to use the group abstraction. A group is a collection of objects that appear to other objects interacting with it, to be a single object. As shown in Figure 2, all Fuselet replicas form a *replication group*, which is connected[†] with the guardian that interacts with it.

A general purpose architecture supporting replication, must accommodate replication of stateful as well as stateless objects. Replicating a stateful application object under the group abstraction, as done in the ITUA architecture imposes two requirements on the application objects being replicated. First, the application object must support an interface for exchanging state information so that lower-level mechanisms can maintain the group abstraction. Second, as in any other state-machine-based [34] replication approach, the replicated application object must be deterministic.

Managers

For effective coordination among redundant hosts and application objects (some of which are replicated) on these hosts, a management agent is needed in each host. In the ITUA architecture the management object, denoted as the manager, fulfills that role. A manager M on a host h integrates the various ITUA defense mechanisms relevant for h into a single entity that allows h to be a part of the architecture. Collectively, the managers (and hence the *manager group*[‡] abstraction in Figure 2) are responsible for executing a number of the defensive adaptations.

As shown in Figure 2, a manager is composed of two major functional components namely, the Security Mgr and the Replication Mgr. The latter is responsible for managing replicas, while the former is responsible for responding to security incidents in individual hosts, as well as managing the security domains. Both these functions gather and aggregate information that

[†]The *connection group* [31] is often used to represent the connection abstraction.

[‡]The present implementation of the ITUA architecture has all managers in the single manager group. This may not scale well: if the number of hosts become large, time required for a communication round within the group may become unacceptably long. Alternatives, such as a hierarchical organization of managers are possible. However, our experiments with the flat singleton manager group showed that the communication delay was acceptable for the IEIST application.

is important for effective intrusion tolerance and needs to be shared between the two functions across the manager group. To facilitate this sharing in a disciplined way, dissemination of such information in the ITUA architecture is always done through the Replication Mgr (as the arrows indicate). Defensive responses that are local to the manager's host, such as blocking suspicious network traffic entering the host or recovering a file, are mounted by the sensor-actuator pairs under the Security Mgr function without any coordination with other managers. More sophisticated defensive responses, such as starting and killing replicas or isolating a security domain are mounted as a result of a wider coordination within the manager group.

Intrusion-Tolerant Gateways and Group Communication System (GCS)

It is better to build the communication and voting algorithms needed for Byzantine fault tolerance in a reusable manner separating it from the application's functional implementation, than to encode them within the application code. This separation of concern however, requires an application level interface. Since ITUA is designed for distributed objects applications, this interface is in the form of an object gateway. The gateway element in the ITUA architecture encapsulates the complexities associated with Byzantine fault-tolerant communication and voting, and acts as a proxy for remote objects. Each member of a replication group (e.g., a Fuselet replica) uses a gateway to communicate with other members of the replication group, as does an application object that interacts with a replicated object via a connection group. Similarly, each manager uses a gateway for multicasting messages to other managers in the manager group. Internally, the communication and membership protocols supporting the group abstraction are built at the GCS layer assuming the existence of an IP infrastructure for transporting packets. The voting protocols implemented at the gateway on the other hand, assume the existence of a GCS layer below.

Organizing Adaptation for Intrusion Tolerance in the Middleware

Figure 3 shows three system layers, with the application layer on top, the OS and infrastructure layer at the bottom, and the middleware layer in between. The defensive adaptations desired for intrusion tolerance include changes in the application-level behavior (e.g., using Byzantine fault-tolerant Fuselet replicas vs a stand-alone Fuselet) as well as manipulation of the infrastructure layer resources (e.g., isolating a corrupt host). We argue therefore, that middleware is the most suitable place to organize such adaptive behavior since it allows the adaptation mechanism(s) to have access to the application layer above and the infrastructure layer below. As shown in Figure 3, the mechanisms that are responsible for defensive adaptation in ITUA are designed as middleware services and constructs. Within the middleware layer, the mechanisms themselves are organized into multiple layers: starting with the GCS at the bottom, followed by the gateways in the middle and managers and QuO contracts [40] on top.

The placement of the gateway and the GCS in the architecture allows introduction of two levels of defense against an attacker's attempt to corrupt application objects. The ITUA GCS provides a process group abstraction, and ensures that all correct processes in the group have the same membership and deliver the same set of messages in the same order. In doing so, the

GCS enables tolerance of integrity attacks at the communication-stream level. The gateways are built on the substrate provided by the GCS. Complementing the GCS, the gateways defend against integrity attacks at the object-interaction level. Higher level abstractions such as the replication and manager groups are built using the gateways.

Apart from managing replicas, a manager uses infrastructure-level resource controllers to mount adaptive responses. As mentioned earlier, some of these responses are mounted under local (i.e., its own) control, while others are mounted under the collective control of the manager group. In addition, the connection with application objects via the QuO (Quality Objects) contracts allows the possibility of an application object interacting with the resource controllers. In all these cases, the managers essentially provide a *middleware service* that is used in defending the application.

The middleware construct known as the QuO contract, is mainly used to intercept and modify the application object's interaction with other application objects as described in [40]. As explained above, the QuO contract can interface with the managers as well. This makes the manager aware of application level events, and enables the application to influence management decisions. Participation of all application objects in defensive adaptation may not be necessary in all situations. For instance, in the IEIST example, the guardians need to adapt from using the replicated Fuselets to using the stand-alone Fuselets. But a stand-alone Fuselet does not need to become a Fuselet replica (or vice versa). In Figure 3, the QuO contract is shown in dotted lines to indicate that not all application objects are required to use a QuO contract in the ITUA architecture.

The remainder of this section presents more details on the realization and defensive capabilities of the four mechanisms that are organized in the middleware layer of the ITUA architecture[§].

Group Communication System (GCS)

The GCS implements a protocol stack that guarantees certain properties despite communication from some (less than a third of the all) group members appearing to be corrupt. One such stack is instantiated for each group an object belongs to (in Figure 2, the Fuselet replica has a separate GCS for the replication and connection groups). There are three key intrusion-tolerant protocols in the ITUA GCS stack that we will highlight here: the Group Membership Protocol (GM), the Reliable Multicast Protocol (RM), and the Total Ordering (TO) protocol. The protocols are based on the timed asynchronous system model [5], which circumvents the impossibility of consensus in an asynchronous environment [6] by defining time-outs for message transmission delays and process scheduling delays based on each group member's local hardware clock. The local hardware clock of the member processes need not be synchronized with the clock of other processes, but it is assumed that all the local clocks proceed within a linear envelope of real time. The protocols employ standard cryptographic mechanisms based on public-key cryptography, such as message digests and digital signatures,

[§]Implementation details such as interfaces and class descriptions, along with the ITUA release are available on request from the authors.

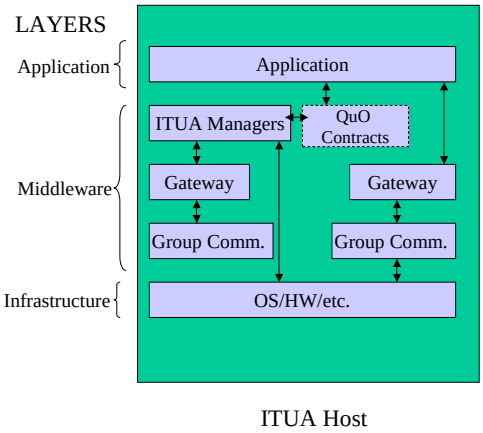

Figure 3. Adaptive Intrusion Tolerance Support in the Middleware

to prevent spoofing and replays. Here, we briefly describe the three protocols (see [**?**] for details).

The GM protocol is responsible for maintaining consistent group membership information across all correct member processes, removing processes from the group, and adding new processes to the group. The protocol installs a series of views, $V_0, V_1, \cdots$. A *view* is a set of identifiers of processes that are members of the view. A view change is necessary when the group membership changes, i.e., when members leave or join. The processes in a single view V have integer ranks from 0 to $|V| - 1$. Hence, the group members corresponding to the view V are $p_0, p_1, \cdots, p_{|V|-1}$. When a new view is *installed*, the lowest-ranked process in the view, p_0, becomes the *leader* of the view. The leader has no special privileges, but is expected to initiate the next view installation when "enough" other members demand it (when a majority of correct processes suspect a member and want to exclude it from the group, or when a majority of correct processes want to include a new member in the group). If the leader is suspected by enough members to be corrupt, then the next-lowest-ranked process initiates a view change, and so on. The GM protocol ensures that all correct processes see the sequence of view changes in the same order.

The RM protocol guarantees that all correct processes deliver[¶] the same set of multicast messages. This is an important property required in the ITUA architecture. For example, a manager might want to inform the manager group about the infiltration of a domain; all managers would need to receive this message. The RM protocol also ensures that the contents

[¶]We use the term to mean that processes at the GCS level deliver the message to the application that is using the GCS. In other words, this "delivery" happens when the message is received by all correct processes at the GCS level.

of a particular multicast message (i.e., a particular sequence number) as delivered at all correct processes are the same. It thus prevents situations in which a malicious manager sends a correct report to some subset of the correct managers and a false report to the rest of the correct managers. The RM protocol takes the common techniques of message buffering, sequence numbers, and positive and negative ACKs to address the problems of messages being lost, ordered, or delayed. It uses message digests and digital signatures to ensure that a message is delivered to all correct processes without any change in its content, even in the presence of corrupt senders. When a message needs to be transmitted, the sender buffers a copy of the message, creates a signed digest for the message, and sends the digest to the group. On receiving such a message, the multicast protocol creates a signed reply to it, and sends the reply back to the sender of the digest. The digests sender collects replies from two-thirds of the group members and then sends out the actual message with the signed replies from two-thirds of the group members attached. On receiving such an authenticated message, the protocol checks the validity of the attached signatures and accepts the message.

In the ITUA architecture, all members of a replication group must maintain the same internal states. For that purpose, they should all deliver the same set of multicast messages in the same order. The TO protocol guarantees this property. Our TO protocol is an adaptation of the "born-order" protocol [2] for total ordering, in which the messages contain information about the order in which they should be delivered. All TO protocol messages are multicast using the RM protocol. The TO protocol achieves total order by assigning sets of global sequence numbers to the group members at the time of view installation. Individual sequence numbers are assigned by processes to messages they multicast, and messages are delivered in the order of the sequence numbers by all processes. A faulty process can stall the progress of the TO protocol by not sending a message. We avoid this by forcing group members to transmit protocol-level null messages (i.e., no payload) if they do not have any other messages to send. All processes monitor the progress of the protocol. A silent process, i.e., one that does not even send null messages, will be declared faulty and reported to the GM protocol, which will then remove the faulty process from the group.

Intrusion-Tolerant Gateway

The intrusion-tolerant gateways provide a robust, reliable, and flexible mechanism for application objects to interact transparently within a framework of groups. A simplified top-down description (more details can be found in [35]) of the extensible gateway construct is as follows. First, the gateway exports an interface (CORBA or socket) to receive and deliver messages from/to the local application-level entity (i.e., an object or a process). It is the gateways' responsibility to transparently manage the communication of the local application-level entity's messages to a recipient entity that is possibly replicated and to allow only a single response to reach the local entity. Second, there is a choice of one or more *handlers* responsible for the intrusion-tolerant interaction with remote, possibly replicated, entities. The handlers send and receive the local application-level entity's invocations to and from the remote entity through the group communication system. Third, a state transfer processor implements a protocol for consensus among replicas on each state transferred and installed in new replicas. Fourth, the Alert Sender component is used to notify the managers of faults detected in

other gateways (wrong proof, signature, crash or voting failures, and so forth.). Finally, at the bottom, there is a GCS adapter layer supporting pluggable integration with a specific GCS for inter-gateway communication. The specific configuration of gateway components, such as the handlers and adapters, is specified by a configuration file and is loaded from dynamically linked libraries at startup time. New handlers and GCS adapters can be easily developed as dynamic libraries and interfaced to the gateway. Different handlers implement different replication schemes and communication schemes for different purposes, such as fault tolerance or timeliness/consistency trade-offs or intrusion tolerance. In the context of intrusion tolerance, we make use of two handlers: 1) a multicast handler, which provides reliable atomic multicast [3] for messages sent by a manager to other managers, and 2) the SBMV (sender-based majority voter) handler [15], used by replicated processes, which provides tolerance against Byzantine behavior of a replica. The SBMV handler was developed specifically for the purpose of tolerating corrupt interactions in the group framework (among the replicas or between a client and a replication group).

The ITUA Managers

The collective functionality of the managers revolves around two objectives: 1) managing adaptive responses related to replicated objects, and 2) managing adaptive responses to security incidents and changes in trust relationships.

Replication Management

Application objects under the ITUA architecture are started by managers at the initialization time. While the application runs, ongoing adaptation determines which objects will be started or killed, when, and on which security domain or host. The following sections describe the capabilities that the managers must have in order to support this kind of adaptation, and the algorithms used to implement these capabilities.

Replica Kill The ITUA GCS and the gateways enable ignoring inputs from a faulty replica as long as there are enough ($3f + 1$ when f is the number of faulty replicas) correct replicas. However, a faulty (in particular, corrupt) replica left alive can be used as a launching pad to attack other parts of the system. To balance this risk with the risk of the attacker abusing the replica kill mechanism, the ITUA architecture allows killing of a replica under the following conditions: 1) a replica r can only be killed by the manager M that started it, and 2) M will kill r only if a majority of correct managers (i.e., they are not suspected by M) report r as faulty. However, the manager M itself may be corrupt, and therefore may choose not to kill r, or to kill it when it is not faulty. The managers are designed to mount a coordinated response by isolating the manager M and its host under these circumstances.

Replica Start The managers use a probabilistic approach to starting new replicas that introduces a level of uncertainty for the attacker. The approach also ensures that no faulty replica or manager can control the placement of the new replica.

Any manager in a security domain that does not already have a member of the replication group R is *eligible* to start a new member of R. The probability that such a manager will start a replica is $1/m$ by design, where m is the total number of managers in security domains that do not have members of R. This is ensured as follows. When a new replica is needed, each eligible manager will compute the value of $1/m$ (m is part of the global state maintained by the distributed managers) and choose a random number (between 0 and 1). If the random number is smaller than $1/m$, the manager will notify the other managers of its intent to start a new replica, start the replica, and commence the dynamic keying process (described later) to facilitate acceptance of the new replica in the existing replication group. Note that under this scheme, it is possible that none of the eligible managers or multiple eligible managers may choose to start a replica.

When replicas need to be replenished, each manager listens for other managers' indications that they intend to start new replicas. If, after some time (determined by our timed asynchrony assumption), not enough replica starts have been proposed, eligible managers will repeat the above steps. If there are enough correct managers, eventually enough replicas will be started.

While this algorithm does not guarantee that exactly one new replica will be started when one is needed, its key properties are bounded within reasonable limits by design. For instance, the probability of no manager choosing itself in a given round is: $p = (1 - \frac{1}{m})^m$. As m goes to infinity (i.e., the limit for large groups), p goes to $\frac{1}{e}$. This value is close to .37, but as noted before, if no replica start proposals are recieved within a predefined time, managers initiate a second round of choosing candidates for replica start. The expected number of rounds until there is a round in which at least one manager chooses itself is $\frac{1}{(1-p)}$, which is between 1 and 2 for all values of m and goes to $\frac{1}{1-\frac{1}{e}}$ (approximately 1.56) as m goes to infinity. The probability that exactly 1 manager will choose itself[‖] is $q = (1 - \frac{1}{m})^{m-1}$ or $q = \frac{pm}{m-1}$. Although p is smaller than q, q also goes to $1/e$ as m goes to infinity.

It is also possible that more replicas than necessary may be started (with probability $1-p-q$). The architecture forbids multiple replicas in the same security domain, but otherwise allows extra replicas to be started. If more than one manager in a domain proposes to start a replica, the managers will support only the first message they receive from that domain. However, eligible managers in different domains can choose to start replicas at the same time. It is possible to avoid extra replicas by supporting only the first start proposal, but this introduces a dependency on the ordering of messages that could be exploited by a malicious manager.

A replica start involves physically creating the process that runs the object from executables stored (and monitored by ITUA Security Management function) on disk. After the new object is authenticated and admitted to the group by existing group members (see Dynamic Keying later in this Section), the new object obtains the state from existing replicas. This, and the fact that the new replica is started on a different security domain from the domain of the compromised one, minimize the risk of attack propagation through replica starts.

[‖]Probability q can be obtained by summing the probability of a specific manager choosing itself i.e., $\frac{1}{m} \times (1 - \frac{1}{m})^{m-1}$ over all managers.

Replica Prestart As shown later in this paper (see the section entitled *Performance Evaluation*), the full replica start algorithm takes of the order of 10 seconds to complete, and is susceptible to CPU load. To minimize the delay involved in the probabilistic rounds of the replica start algorithm, the managers are designed to support replica prestarts. Under this design, a number of replicas are started, but are not allowed to join their groups. This is accomplished by performing only the first step of the dynamic keying algorithm and not immediately distributing the "new" replica's public key. When a new replica is required, the manager has a replica ready, and only needs to distribute the key. The application's survivability strategy usually determines whether prestarting, as opposed to on-demand starting, is required.

Dynamic Keying In order for the new replica to be a legitimate part of the application, our architecture requires that its public key must be known to all parties that interact with it. Rather than pre-distributing key pairs, the new key pairs are generated when a new replica is started. This "Dynamic Keying" ensures that only the replicas started by trusted managers can join the group of existing replicas. By coordinating the acceptance of new keys among the existing replicas, the managers can exclude replicas that are untrusted. Dynamic keying has two steps: generation and distribution of the keys.

Key generation is done by a collaboration between the newly started replica, r, and its manager, M. When r is first started, M passes r a symmetric session key on its standard input. Then, r generates a key-pair and sends back the public key encrypted by the session key. Use of the session key ensures that the public key, which was received via a CORBA call at M, actually comes from r and has not been tampered with on its way to M. Once M receives r's key, it generates a certificate associating the public key with r, and multicasts the certificate to the other managers. The second step of the dynamic keying algorithm begins when all managers, including M, receive the certificate for the new replica. At that time, all managers that have members of R (r's replication group) examine their quarantines (described in the next section) to check whether M should not be trusted. If the consensus is that M is not in quarantine, these managers will then send r's certificate to their replicas in R. Even if some correct manager does not agree with the consensus, it follows the decision of the group. When M receives the certificate it multicast, it sends the certificate to r, along with the certificates for the other replicas in R that M collected when those replicas were started. When r receives a certificate with its key, it will then attempt to join its group. If the managers accept r by giving r's certificate to other members of R, then r will be admitted; otherwise, r will not be able to interact with the other replicas and will be isolated.

Management of Security and Trust

In this section, we describe how the managers respond to low-level security incidents locally and how this response combined with information about corrupt replica or manager behavior, leads to changes in trust dynamics among the managers. As mentioned earlier, inter-manager coordination in starting or killing replicas is influenced by the level of distrust among the managers. To contain an attack in the extreme case, managers may collectively decide that an

entire security domain cannot be trusted anymore, and therefore mount an adaptive response to isolate that domain.

Rapid Reaction Loops The first line of defense in an intrusion-tolerant architecture is usually mounting a quick response to (early) attack actions. In the ITUA architecture, this type of rapid reactions are implemented in the rapid reaction loops. A manager can have multiple such loops, where each loop combines a sensor (which is usually an Intrusion Detection System) and an actuator that executes the knee-jerk reaction to observed events. In addition to mounting a quick response, the loops in the ITUA architecture monitor how effective the response was, whether the same attack was repeated, and whether multiple attacks were launched in parallel. It sorts and prioritizes the monitored information and produces a summary report for the manager group.

We will describe two such loops that we have experimented within the ITUA architecture. The *portattack* loop couples the Snort [32] network IDS with the Netfilter [19] Linux firewall. Upon detection of network-based malicious activities such as port scans, this loop dynamically reconfigures the firewall to block traffic from offending sources. The loops are designed to support the notion of white and black lists: hosts that should never be blocked can be specified in a white list and hosts that should always be blocked can be specified in a black list. The *filesystem* loop combines the Tripwire [11] file integrity checker with a backup utility. Upon detection of an unauthorized change in the file system, the loop restores corrupted files from a secure backup (i.e., CD).

The loops also introduce a level of unpredictability in the adaptive response they mount. In the portattack loop, for instance, the type (e.g., ICMP reject or silent dropping of packets) or duration of port-blocking are chosen with some randomness, making prediction hard or impossible for the attacker. Similarly, in the filesystem loop, the hashing algorithm or frequency of file scans and the set of files** that are retrieved from the secure backup are chosen unpredictably.

Filtering and Information Reduction The amount of security information gathered in a manager M is quite large. The sources of such information include the rapid reaction loops as well as the replicas that are running on M's host. Therefore, managers go through a process of information reduction before exchanging security information with peer managers. Low-level alerts obtained from loop sensors are combined into loop-level alerts and further categorized into high, medium, or low severity. Managers exchange the alerts with each other using the GCS. The managers give high-severity incident reports higher priority than reports with lower severity. Loop-level alerts about a manager M play a role in deciding whether other managers should trust M (i.e., whether M is in quarantine: see below) as well as whether they should trust error reports about M that are coming from the replication groups (such as a report that M's replica has just been corrupted) or other managers (such as a report that M did not respond when expected to). However, the managers give more weight to the error reports

**This set must include the damaged files, but usually a larger set is retrieved to camouflage the repair and to make it look like a routine synchronization activity.

than to the loop-level alerts, since the loop-level alerts are coming from a single source, and the accuracy of intrusion detection systems is questionable.

Quarantine Even though the managers use consensus about observed events before mounting an adaptive response, there is some level of trust among the managers. Observed behavior, loop-level alerts, and error reports from the replication groups alter that trust relationship. *Quarantine* is a concept used in the ITUA architecture to capture this dynamic manipulation of trust. Quarantine is a system-wide notion, but each correct manager enforces it locally. When a manager M_1 decides that it no longer trusts (some examples of situations in which managers may decide they no longer trust other managers are presented earlier in this section) a fellow manager M_2, this decision is atomically multicast to the other managers. That ensures that 1) all the correct managers become aware of M_1's view of M_2 and 2) each receiving manager receives the messages in the same order. If M_1 later decides to remove M_2 from quarantine and trust it again, that decision is also atomically multicast to the other managers. When a manager receives a quarantine message, it updates a table that stores each manager's view of every other manager. A manager can look at the table to get an indication of the trustworthiness of a fellow manager. If a manager M is in quarantine in the view of most managers, M's capability in the architecture is severely restricted. For instance, the replicas started by it will not be admitted. Greater suspicion of M will increase the time M remains in quarantine.

Quarantine is another source for unpredictability in ITUA adaptive responses. When M_1 has assigned M_2 to quarantine, the amount of time M_1 needs to observe M_2 before it can consider removing M_2 from quarantine can be varied. That makes it harder for the attacker to find out how long M_2 will be in quarantine.

Domain and Host Isolation Quarantine is meant for temporary suspension of trust based on observed behavior. However, there are situations in which more drastic action, such as isolation of a single host or an entire security domain, is called for. The ITUA architecture can accommodate such actions, because it has redundant security domains. Upon such isolation, the isolated host (domain) is essentially removed from the architecture, it can no longer be a platform to host application processes, and its manager can no longer participate in the system-wide coordination. The local firewalls enforce this isolation by rejecting traffic to and from the isolated host (domain). The loss of an existing application-level object on the host (domain) is compensated for by the starting of a new object on other hosts (domains). The strategy that decides when to isolate a host or a domain is tunable and application dependent. A simple strategy we experimented with singleton domains (i.e., each domain has a single host) can be described as the "one-strike policy," in which a host h is isolated if the managers agree that either a replica or the manager on h is corrupt. Under this policy, the above automatically implies isolation of the domain that contained the isolated host as well.

Supporting Application Level Adaptation - Graceful Degradation

The primary motivation behind using application level adaptation is to support graceful degradation, i.e., the desire to allow the application to operate in a degraded mode when

a partially successful attack has compromised a significant portion of the available resources. The ITUA architecture accommodates application level adaptation by weaving self-monitoring code into the application itself using the QuO adaptive middleware [14].

Effective graceful degradation is usually application specific[††], however there is one generic case in the architecture that we experimented with, and introduced earlier in terms of the IEIST example. When multiple security domains are compromised and it is not possible to maintain the required number of replicas anymore, the application should adapt to give up Byzantine fault tolerance, and start using individual stand-alone objects. To support this, a small number of stand-alone "back-end" server instances that have offline and lazy methods of synchronization, can be maintained (not used to serve actively until this point) in hardened hosts. Note that this degraded mode of operation will not tolerate corruption of the back-end servers, and some interactions may be based on outdated information (if there is a lack of tight synchrony among the back-end servers).

Unpredictability can be useful in application-level adaptive responses. If there are more than one back-end servers, the application can select among them randomly for a given interaction. If there is a single back-end server, the application may hide the real interaction among a set of "decoy" interactions. The QuO middleware used in the ITUA architecture supports injection of this kind of uncertainty.

Discussion of Experience in Using ITUA and Lessons Learned

In this section, we first describe our experience in using the ITUA architecture for demonstrating adaptive defense in a realistic military context. We then discuss the lessons learned in the process. We conclude this section with a short summary, positioning our work in the long term vision of cyber-defense.

Using ITUA in Defending an IEIST Application

We have demonstrated the use of our technology in defending a military application on multiple occasions, including DISCEX '03 [24] and DarpaTech 2002. The demonstrations were based on the IEIST concept introduced earlier, and used several real IEIST components. Multiple fighters and UCAVs are simulated in the demonstration prototype by repeated registration of the fighter and UCAV guardians (GAs) with the Fuselet. Since our focus is on continued availability and correctness of service provided by the Fuselet, the Fuselet is replicated in four security domains (for tolerating one Byzantine corruption), where each domain is a single host. Three additional single-host domains are added, two to run stand-alone non-replicated

[††]For instance, when Byzantine fault-tolerant replication of the Fuselet is untenable, all the guardians could form a logical ring among themselves. Each UCAV puts the threat information it observes through its guardian in that ring. The threat information circulates through the ring and the fighter looking for that information can eventually find it. In another project [25], we have demonstrated an implementation of a similar graceful degradation.

versions of the Fuselet and a third to run the guardians. The survivability strategy adopted for experimenting with this demonstration prototype is as follows. Starting with four replicas of the Fuselet, the architecture attempts to maintain them by replenishing them as necessary at unpredictable locations throughout the seven domains. A domain is isolated if either its replica or manager is corrupted. If severe loop-level alerts or replica corruptions are reported simultaneously at three domains, the application switches to using the stand-alone Fuselets. The demonstration prototype was subjected to various attack-steps and attack-effect injections in a series of experiments. The following is a summary of the range of defensive adaptations the defense-enabled demonstration application was able to mount in response.

Range of Attacks and Defensive Responses

Early attack steps such as *Reconnaissance* by scans[†] are detected by the portattack loop, which blocks the source of the scan, making it harder for the attacker to get the desired information. Assuming that a sophisticated attacker will overcome[‡] this first line of defense and infiltrate a host, the next stage of our experiment shows how the defense-enabled system responds to attacker's attempt to corrupt mission-critical files (such as executables and dlls). This is picked up by the file system loop, which restores the damaged file(s) from a CD-ROM backup. These two rapid-reaction defensive adaptations are mounted without any coordination with other parts of the system. Their scope is very local (e.g., within the host), however, the managers note their occurrence.

In the next stage, we emulate an attacker in control of a Fuselet replica running on an infiltrated host. Malicious behavior of the hijacked replica (such as sending extra messages, sending no messages, or sending corrupt messages) is detected by the Byzantine fault-tolerant replication group, and the three other Fuselet replicas collectively provide correct service to the fighter and the UCAV GAs. In parallel, the faulty replica is reported to the managers.

After observing the Fuselet replicas survive the corruption, the attacker is expected to attack the gateway and the GCS layer that contributed to the Byzantine fault tolerance. Consequently, we allow corruption of the gateway and the GCS underneath the hijacked Fuselet replica in the next stage of our experiment. This results in the hijacked replica looking faulty to the rest of the Fuselet replicas. The other replicas continue to provide correct service, but now that the corruption is observed at the gateway layer, the managers coordinate among themselves to replace the faulty gateway and the replica attached to it.

Following the principle of no implicit trust, the managers in the ITUA architecture actually monitor the system for the results of their collective decision. They can detect if a manager that was instructed to kill or start a replica did not do so, and mount a response such as putting the offending manager in quarantine, isolating the security domain, or starting other

[†]This does not consider reconnaissance involving passive scans, for which there is no observable impact on the defense-enabled system. In another context, we have explored proactive adaptation that changes the configuration of the defended application over time as a means to defend against such threats [39].

[‡]For instance, if the attacker spoofs the source of his scans to be in the white list of the loop, such scans will go unblocked. As we have explored in [25], anti-spoofing techniques that authenticate source addresses (as in a virtual private group) can be used in the architecture to harden this.

replicas. In the next stage of our experiment, the attacker makes the manager of the infiltrated host start a number of replicas running attack code. Even if the attacker steals the keys from existing replicas on the infiltrated host, the dynamic keying algorithm prevents the admission of these new replicas into the existing replication group. Furthermore, this unauthorized act is significant enough, that the other managers collectively decide to isolate the infiltrated domain and its manager.

The next stage escalates the attack further by assuming that the attacker has infiltrated multiple security domains and corrupts multiple replicas. Since we started with four Fuselet replicas, the defense-enabled application will not be able to provide Byzantine fault tolerance in the presence of more than one simultaneous corruption. This results in an application-level decision to abandon use of the replicated Fuselet. In this degraded mode, the stand-alone replicas are used, and the system sacrifices Byzantine fault tolerance. Adaptive middleware code (QuO Contract in Figure 3) at the guardians will pick randomly between the two stand-alone Fuselets to confuse the attacker in his attempts to infiltrate them.

Lessons Learned

The demonstration prototype is clearly a proof of the concept that auto-adaptive intrusion-tolerant applications are realizable. Apart from establishing the technical feasibility by experimenting with the prototype, we also performed a comprehensive evaluation of our technology as reported in [8, 10, 30]. The conclusion that we draw from these efforts is that this is a big step forward in the right direction, but there is a need for more work to make the technology ready and attractive for transition to fielded systems. To summarize the results of our evaluation, and to show how we reached the above conclusion, let us try to answer the following two questions:

1. Was our technology effective in achieving intrusion tolerance?
2. Was the performance cost of defense-enabling acceptable?

Effectiveness of the Technology

Experiments with the prototype, and analysis of the results show that the most fatal situation occurs when the attacker infiltrates a significant number of security domains *undetected* and *unchallenged* by mechanisms within the ITUA architecture and introduces simultaneous corruption. Under this circumstance, the attacker has essentially forced the application to adapt to the degraded mode. When stand-alone Fuselets are being used, the attacker simply needs to find them and kill or corrupt them. The fact that the attacker has to infiltrate multiple security domains, remain invisible and bypass our adaptive responses, and also defeat the application adapting to provide a degraded service, shows a significant increase in attacker's level of difficulty, and therefore, a significant improvement in the application's overall survivability.

Our analysis shows that the enforcement of the security domains play a crucial role in increasing the applications chances of survival. The defense stands to gain a lot by grounding this enforcement in the hardware and operating system level that are hard to bypass or

circumvent. Examples of techniques that can be used in this regard include using cryptography-based firewalls on network interface cards (NICs) [23], limiting the cryptographic keys within smart cards and not allowing them in the CPU or main memory, or using domain enforcement as in SELinux [22]. The defense will benefit further if the operating environment has physical diversity, for instance, diverse host architectures, operating systems or other runtime entities such as the Java Virtual Machines (JVM) if applicable. We are currently experimenting with architectures that implement these ideas.

Another issue that is often raised about adaptive intrusion tolerance involves the potential dependency of defensive adaptations on intrusion detection. In our experiments and demonstrations we make use of attack tools, such as Nessus [18] and nmap [21], as well as fault injections to simulate the attack effects at various stages. It is important to note that when fault injection is used to simulate an attack effect, it signifies that the attacker has managed to infiltrate the host *without being detected*. The fact that the defense-enabled system is able to survive such injections shows that our technology is not crucially dependent on accurate detection of intruders. However, availability of better and more accurate intrusion detections will make our adaptive architecture stronger.

Performance Evaluation

The gateway and the GCS are used by both application objects and managers, and hence are important factors in determining the performance cost of the ITUA architecture. The results of our performance evaluation studies [7, 15, 29] show that the most significant contributor to the performance cost of intrusion tolerance is cryptography, and this contributor was especially significant under loaded conditions and larger group sizes. This leads us to believe that the overall cost can be reduced significantly through the use of specialized cryptographic hardware and faster machines. Such hardware is becoming more readily available and affordable with time, and it is reasonable to assume that mission-critical systems such as IEIST would be equipped with such hardware. Essentially, there is a tradeoff between the level of tolerance and the performance cost: more tolerance will require more replicas in more protection domains, and even with the availability of faster machines and specialized cryptographic co-processors different applications will choose different tradeoffs.

Our results also show that the cost of removing members from the group depends on the fault detection mechanisms (detection based on timers are generally slower than detection based on message patterns and contents).

In the remainder of this section, we evaluate two other key factors in determining the performance cost of the ITUA architecture namely, 1) response time of the rapid reaction loops, and 2) response time of the managers.

Rapid-Reaction Response Time In our experiments [9] with the demonstration prototype, the average time taken by the Snort-based sensor mechanism of the portattack loop to detect a portscan was 13630 milliseconds with a standard deviation of 384 milliseconds. This measurement was done under unloaded condition, and included the time taken by Snort to detect the portscan and log the packet information plus the time it took to analyze the log file and send an attack notice to the Netfilter-based actuator. For this test Snort was configured

with 760 unoptimized rules, and the host running the loop was a low range (600 Mhz) desktop. The actuator response (from the time it received the attack notice to actually blocking the IP address) took on the average 540.3 milliseconds with a standard deviation of 282.9 milliseconds. Most of the delay in the sensor part was due to the file based interface between Snort and the portattack loop software. Similar experiments with the filesystem loop show that the mean time taken by the Tripwire-based sensor to detect a file removal was 8574 milliseconds with a standard deviation of 3880 milliseconds under unloaded condition. In these experiments, an executable file of size 1.3 MBytes used by the implementation was removed. The mean time it took for the file to be replaced was 2746 milliseconds with a standard deviation of 1661 milliseconds. Further experiments show that the detection time is more susceptible to stress (network load for portattack, and CPU load for filesystem) than the response time for both the loops. Smaller set of Snort rules with an interface that is not file-based is expected to improve the performance of the portattack loop. Even though the filesystem loop has to compute hashes of files, avoiding file-based interface for communication to and from Tripwire is expected to improve its performance as well.

Manager response time Experimenting with the demonstration prototype in a configuration where each host is its own security domain, we measured the time taken by the managers to start a new replica. This gave us an estimate of the cost of the probabilistic replica start algorithm. The hosts used in this configuration were similar to those mentioned above (low range desktops), connected in a LAN[§]. The results show that the starting of a replica (from the time a replica is killed to the time a replacement has successfully joined the replication group) takes on average, 16250 milliseconds with a standard deviation of 2872 milliseconds under unloaded condition. This estimate includes the time to detect the death of a replica, and probabilistic rounds involving multicasts over the GCS as well as the cost of cryptographic operations (key generation, and signing). While this average startup time was acceptable for the IEIST application − as long as there were enough other replicas, there was no noticeable disruption in service − further experiments revealed another important issue. It was observed that under heavy stress (CPU load), the probabilistic replica start algorithm did not terminate in many runs. This was one of the motivations behind incorporating the support for replica prestarts. By prestarting a pool of replicas, and executing the key exchange when a new replica was needed, this problem was significantly reduced.

Summary

Experience shows that perfect protection from or accurate detection of cyber-attacks is unachievable in practice. Hence, an architectural approach that combines protection, detection and adaptation in order to survive the attack-effects seems to be the only effective strategy against cyber-attacks. We believe that the ITUA architecture is a successful first step in

[§]To make the network realistic for the IEIST scenario, the links in the LAN were configured with varying capacity with one high (100 Mbps) and one low (2 Mbps) capacity link and the remaining links being medium capacity (10Mbps).

that direction. We have created an integrated architectural approach for developing auto-adaptive intrusion-tolerant applications, and used the technology to defend a critical avionics application. The technology is expected to mature as we continue to investigate and experiment with the architecture, providing higher levels of effectiveness and performance, wider attack coverage, and offer usability and cost-benefit trade off.

Conclusion

In developing and using the ITUA architecture, we have shown that it is possible to combine redundancy, Byzantine fault tolerance, and adaptive responses in an integrated architecture to provide tolerance against malicious intrusions. We have demonstrated that a range of autonomic adaptations, based on these techniques and organized at the middleware layer, can increase an application's ability to remain operational despite corruption at various parts of the system. Under a carefully designed defense strategy, autonomic adaptation facilitate isolation of compromised resources, failure recovery, and graceful degradation.

The primary contributions of this paper are 1) an architectural approach to develope auto-adaptive intrusion-tolerant applications as exemplified in the ITUA architecture, and 2) the experience gained and lessons learned from our efforts to use the architectural approach in defending a realistic application. Other contributions include techniques and mechanisms for 1) using Byzantine fault-tolerant replication in a dynamic way where corrupt replicas are removed and new replicas started, 2) using graceful degradation when Byzantine fault tolerance is untenable, 3) managing a redundant number of hosts organized in security domains in a decentralized manner without implicit trust, 4) injecting uncertainty in adaptive responses, and 5) integrating multiple types of defensive responses and adaptation strategies with the application.

REFERENCES

1. A. Adelsbach et al. Conceptual model and architecture of MAFTIA. Technical Report DI/FCUL TR-03-01, Dept. Comp. Sci. Univ. Lisbon, February 2003.
2. K. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, 1996.
3. K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, August 1991.
4. D. Corman, T. Herm, and C. Satterthwaite. Transforming legacy systems to obtain information superiority. In *6th ICCRTS, CCRP*, June 2001.
5. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.
6. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):372–382, April 1985.
7. V. Gupta. Intrusion-tolerant state transfer for group communication systems. Master's thesis, University of Illinois, 2003.
8. ITUA Team. ITUA final report. Technical report, BBN Technologies LLC, December 2003.
9. ITUA Team. ITUA test and evaluation report. Technical report, BBN Technologies LLC, December 2003.
10. ITUA Team. ITUA validation report. Technical report, BBN Technologies LLC, December 2003.

11. G. Kim and E. Spafford. The design and implementation of Tripwire: A filesystem integrity checker. In *Proc. 2nd ACM Conf. Computer and Communications Security*, pages 18–29, 1994.
12. J. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, and M. Gertz. The willow architecture: Comprehensive survivability for large-scale distributed applications. In *Supplemental Volume of the 2002 Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, pages C.7.1–C.7.8, Washington, DC, June 2002.
13. J. P. Loyall, P. Pal, R. E. Schantz, and F. Webber. Building adaptive and agile applications using intrusion detection and response. In *Proceedings of the ISOC Network and Distributed Systems Security Conference (NDSS)*, Februrary 2000.
14. J. P Loyall, R. E. Schantz, J. A. Zinky, and D. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Comp.*, pages 43–52, April 1998. Kyoto, Japan.
15. J. P. Lyons. A replication protocol for an intrusion-tolerant system design. Master's thesis, University of Illinois, 2003.
16. W. Nelson, W. Farrell, M. Atighetchi, J. Clem, L. Sudin, M. Shepard, and K. Theriault. APOD experiment 2: Final report. Technical Report Technical Memorandum 1326, BBN Technologies LLC, September 2002.
17. W. Nelson, W. Farrell, M. Atighetchi, S. Kaufman, L. Sudin, M. Shepard, and K. Theriault. APOD experiment 1: Final report. Technical Report Technical Memorandum 1311, BBN Technologies LLC, May 2002.
18. Nessus home page. http://www.nessus.org.
19. Netfilter: firewalling, NAT and packet mangling for Linux 2.4. http://www.netfilter.org.
20. P. G. Neumann and P. A. Porras. Experience with emerald to date. In *Proceedings of the First USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, CA, April 1999.
21. Nmap home page. http://www.insecure.org/nmap/.
22. NSA. http://www.nsa.gov/selinux.
23. R. O'Brien et al. Intrusion tolerance via network layer controls. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX)*, April 2003.
24. P. Pal et al. Demonstrating intrusion tolerance with ITUA. In *DISCEX 2003, Vol. 2, Part 1: Demonstration Abstracts, Organically Assured and Survivable Information Systems (OASIS)*, pages 135–137, April 2003.
25. P. Pal et al. Designing priotection and adaptation into a survivability architecture: Final design. Technical Report Developed for DARPA, BBN Technologies LLC, August 2003.
26. P. Pal, J. Loyall, R. Schantz, J. Zinky, and F. Webber. Open implementation toolkit for building survivable applications. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX)*, January 2000.
27. P. Pal, F. Webber, R. Schantz, M. Atighetchi, and J. Loyall. Defense enabling using advanced middleware: An example. In *MILCOM*, October 2001.
28. P. Pal, F. Webber, R. Schantz, et al. Survival by defense-enabling. In *New Security Paradigms Workshop*, pages 71–78, September 2001.
29. 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. International Conference on Dependable Systems and Networks (DSN-2002)*, pages 229–238, Washington, DC, June 2002.
30. H.V. Ramasamy, M.Cukier, and W.H. Sanders. Formal verification of an intrusion-tolerant group membership protocol. *IEICE Transactions on Information and Systems, E86-D(12)*, pages 2612–2622, December 2003.
31. Y. (J.) Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. AQuA: An adaptive architecture that provides dependable distributed objects. *Theory and Practice of Object Systems*, 52(1):31–50, jan 2003.
32. M. Roesch. Snort - lightweight intrusion detection for networks. In *USENIX LISA: Thirteenth Systems Administration Conference*, pages 229–238, 1999.
33. D. Samens, B. Matt, B. Niebuhr, G. Tally, B. Whitmore, and D. Bakken. Developing a heterogeneous intrusion tolerant CORBA system. In *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, pages 239–248, Washington, DC, June 2002.
34. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comp. Surv.*, 22(4), December 1990.
35. M. Seri, T. Courtney, M. Cukier, V. Gupta, S. Krishnmamurthy, J. Lyons, H. Ramasamy, J. Ren, and W.H. Sanders. A configurable CORBA gateway for providing adaptable system properties. In *Supplemental Volume of the 2002 International Conference on Dependable Systems And Networks (DSN 2002)*, pages

G.26–G.30, June 2002.
36. S. Singh, M. Cukier, and W. H. Sanders. Probabilistic validation of an intrusion-tolerant replication system. In *Proc. International Conference on Dependable Systems and Networks (DSN 2003)*, pages 615–624, June 2003.
37. United States Air Force Scientific Advisory Board. *Report on Building the Joint Battlespace Infosphere*, 1999. Volume 1: Summary; SAB-TR-99-02.
38. A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou, and T. E. Uribe. An architecture for an adaptive intrusion tolerant server. *Proc. Security Protocols Workshop, LNCS, Springer-Verlag*, 2002.
39. F. Webber, P. Pal, M. Atighetchi, and C. Jones. APOD final report. Technical report, BBN Technologies LLC, October 2002.
40. J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for Quality of Service for CORBA objects. *Theory and Practice of Object Systems*, 1(3):55–73, apr 1997.