A Trustworthiness Detector for Intrusion-Tolerant Group Communication Systems

Soontaree Tanaraksiritavorn and Shivakant Mishra
Department of Computer Science
University of Colorado, Campus Box 0430
Boulder, CO 80309-0430, USA.
Email: {tanaraks|mishras}@cs.colorado.edu
Contact author: Shivakant Mishra

Abstract

A trustworthiness detector in a group communication system raises a suspicion event, whenever one or more group members can no longer be trusted. This suspicion event triggers a group membership protocol to create a new group. This paper describes the design, implementation, and experimental evaluation of a trustworthiness detector that can be incorporated in most group communication systems. The design of this trustworthiness detector is based on two important principles: (1) focusing on observable effects, and (2) detection in depth. This detector is generic in the sense that it is independent of the actual broadcast or group membership protocol being supported by the group communication system. It has been integrated with three different atomic broadcast protocols: a positive acknowledgement-based atomic broadcast protocol, a negative acknowledgement-based atomic broadcast protocol, and a logical ring-based atomic broadcast protocol. The paper describes these implementations, and present an extensive experimental evaluation.

1 Introduction

Modern computing systems are increasingly being used for creating, storing, processing, and transmitting information that is critical to citizens, industry, government, and academia. In fact, it is safe to say that our daily lives are becoming increasingly dependent on a continuous and proper functioning of these systems. As a result, high availability and trustworthiness are the most important requirements of these systems. Despite increased awareness of computer security, incidents of security violations, system failures, and unavailability of important computing services are increasing [18]. With increasing complexity of modern computing systems, and increasing sophistication of security attacks that are routinely launched on these systems, it has become extremely difficult (and impractical) to make these systems completely immune from these attacks. Rather, these systems must be designed to continue to operate correctly even while unforeseen security attacks are successfully launched on them. Intrusion-tolerant systems are systems that remain available and continue to operate correctly despite successful security attacks on parts of the system.

A common technique for building a highly available and intrusion-tolerant service is to replicate the service at multiple components. The key idea is that if one or more service replica are compromised, the rest of the replicas isolate (or shut down) the compromised replica(s), and continue to provide the correct service to the clients. Service replication is typically facilitated by a system-level group communication service that provides support for maintaining the consistency of replication, detecting replica compromise or failure events, and recovering from these events. Examples of intrusion-tolerant group communication systems include [5,7,9,10,12,19,21].

A trustworthiness detector is one of the most important components of an intrusion-tolerant group communication service. This component creates a *suspicion event*, whenever one or more group members cannot be trusted any longer. This suspicion event triggers a group membership protocol to create a new group. Performance of an intrusion-tolerant group communication system critically depends on how well its trustworthiness detector performs. For example, the time it takes to suspect a compromised group member is critical in determining the amount of damage that member may cause.

In this paper, we describe the design, implementation, and experimental evaluation of a trustworthiness detector that can be incorporated into most group communication systems. The design of this trustworthiness detector is based on two important principles: (1) focusing on the observable effects, and (2) detection in depth. This detector is generic in the sense that it is independent of the actual broadcast or group membership protocol being supported by the group communication system. To test the applicability and effectiveness of this detector, we have integrated it with three different atomic broadcast protocols: a positive acknowledgement-based atomic broadcast protocol, a negative acknowledgement-based atomic broadcast protocol, and a logical ring-based atomic broadcast protocol. We describe these implementations and present an extensive experimental evaluation.

This paper makes three important contributions in building intrusion-tolerant systems. First, the trustworthiness detector proposed here has been specifically designed as a separate module for a group communication system. In particular, it does not depend on a specific atomic broadcast or group membership protocol, and exploits the common communication and computation patterns of a group communication system to suspect any possible security compromise of a group member. We are not aware of any trustworthiness detector developed specifically for a group communication system. The existing intrusion-tolerant group communication systems either rely on a generic intrusion detection system [5,7,20,21], or integrate detection process with group membership protocol [12]. By focusing on common communication and computation patterns of group communication systems, the proposed trustworthiness detector can detect conditions that are abnormal from group communication and computation point of view, but appear normal from outside the group. In addition, by logically separating the functionality of trustworthiness detection process from the group membership protocol allows us to integrate this detector in any group communication system.

Second, the proposed trustworthiness detector is based on two important principles: (1) focusing on observable effects, and (2) detection in depth. Both of these principles aid in addressing some important challenges (described in Section 3) that arise in building a trustworthiness detector. By focusing on observable effects, the proposed trustworthiness detector has the potential to detect new unforeseen security attacks, and the detection in depth principle makes it difficult for an adversary to defeat the detection process.

Finally, a prototype of the proposed trustworthiness detector has been built, and the paper provides an extensive experimental evaluation. This is important, because there is a significant lack of any quantitative analysis of the cost of intrusion tolerance at present. There is a critical need of performance data quantifying the cost of providing intrusion tolerance in modern computing systems. The proposed trustworthiness detector has been integrated with three different atomic broadcast protocols, and the paper presents performance evaluation from these implementations. This performance evaluation has provided a very useful insight in detecting malicious behaviors. This is discussed in the paper.

The rest of this paper is organized as follows. Section 2 describes some of the related work in intrusion-tolerant group communication services. Section 3 outlines the challenges in building a trustworthiness detector, and describes the two principles based on which we have built our detector. This section also describes the overall architecture of our trustworthiness detector. Section 4 describes the details of peer-based intrusion detector, which is one part of the proposed trustworthiness detector. Section 5 describes the details of integrating our detector in positive ack, negative ack, and ring-based atomic broadcast protocols, and Section 6 describes the design, implementation and performance evaluation of this integration. Finally, Section 7 concludes the paper.

2 Related Work

Related projects aiming to provide intrusion tolerance at the middleware level using a group communication service include Rampart [20], MAFTIA [21], ITUA [5,7], and SecureRing [12]. In addition, the feasibility issue of implementing a fault detector in an asynchronous distributed system subject to Byzantine faults has been addressed in [1,8,13,14]. The Rampart toolkit [20] provides support for intrusion-tolerant group membership protocol. It assumes that a trustworthiness detector exists to report member suspicions. The trustworthiness detector proposed in this paper can be used for this purpose. ITUA [5,7] uses an adaptive and unpredictable response as a major technique to cope with an attacker. Its architecture separates the role of detection, known as "security advising", from replication management. The security advisor provides traffic information and a quick response to the observed event. Our design on the other hand is finer-grained. In particular, observation of various events, determining if a suspicion event should be raised, and the appropriate response as a result have been explicitly separated. This logical separation is similar to the modularized design of EMERALD [17], and aids in developing adaptive detection mechanisms for varied computing environment and applications.

MAFTIA [21] aims to build a reliable architecture that can cope with intrusions automatically. Three main components of this architecture are dependable middleware, intrusion detection, and dependable trusted third party. The reference model of MAFTIA middleware is divided into two layers: participant level and site level. While the site level copes with physical network, the participant level deals with members engaging in distributed computation. This separation does not suit our requirement in foreseeing the communication and computation of the group communication service itself. The intrusion detection part of MAFTIA offers to combine various intrusion detection results to minimize the number of false alarms. We have incorporated this idea in our project using a mechanism called *output format* described in Section 3.

An earlier work designed to tolerate Byzantine failures on logical ring-based atomic broadcast protocol is the SecureRing protocol [12]. This protocol comprises of three sub-protocols: message delivery, group membership, and fault detection protocols. The main purpose of the SecureRing protocol is to provide reliable ordered message delivery and group membership services despite Byzantine failures. The membership protocol receives information about detectable Byzantine failures from the fault detector and reconfigures the system to form a new membership. The fault detector looks for failure conditions such as mutant messages, improperly formed messages, or notorious actions that blocks the progress of the entire system. In one of our prototype, we have integrated our trustworthiness detector with a similar logical, token-passing ring-based atomic broadcast protocol. Unlike the original protocol that incorporates RSA into the system model to detect failure conditions, our implementation does not explicitly use any cryptosystem in the detection component. Instead, it assumes that the broadcast protocol has access to the cryptography library. This allows our trustworthiness detector to be integrated with any group communication system, while the failure detector of SecureRing is tightly interwined with the ring-based broadcast protocol. Another advantage of our trustworthiness detector is that it does not depend on cryptography to detect failures, although it can take advantage of cryptography if it has been used for implementing a group communication system. It should be noted that a recent study has shown that cryptographic operations contribute significant performance overhead in providing intrusion tolerance [19].

Finally, some work has been done in the area of feasibility of designing a fault detector in an asynchronous distributed system subject to Byzantine faults. It has been shown in [11] that a solution for consensus problem is possible in a *partially synchronous* distributed system subject to Byzantine faults, if and only if the maximum number of faults in the system is less than one-third the total number of processes. A fault detector that detects only *quite* behavior for solving the binary consensus problem in an asynchronous distributed system is proposed in [14]. A process is defined to be quiet if some correct process receives only a finite number of reliable broadcasts from that process in an infinite run. Several fault detectors have been proposed for a Byzantine failure environment in [8]. These detectors are based on the concept of *mute* processes, where a mute process is defined as a process that stops sending protocol messages, but may continue to send other messages.

A classification of Byzantine failure detectors for solving consensus is provided in [13]. Based on completeness and accuracy properties, four new classes of unreliable Byzantine fault detectors have been proposed. Byzantine faults are categorized as detectable and non-detectable faults. Detectable faults consist of omission and commission faults. An omission fault occurs when a message that should have been sent is not sent. On the other hand, a commission fault occurs when a message that should have not been sent is sent. Our trustworthiness detector covers both types of faults by detecting the abnormal behaviors that cause the system to be saturated, blocked, and/or corrupted. The progress of the system is blocked when a message that should have been sent is held back. So, this is an omission fault. The system is saturated when extra non-essential messages are sent and it is corrupted when an ill-formed message is sent. These are commission faults.

3 Architecture

Building a *good* trustworthiness detector for a group communication system is a very difficult task. Some important challenges are:

1. It is not clear what trustworthiness of a component means. What are the conditions under which a trustworthiness detector can conclude that a group member can no longer be trusted?
2. Timeliness of a trustworthiness detector is critical. It needs to raise a suspicion flag as soon as a group member becomes untrustworthy, so that any potential damage that this group member can inflict is minimized.
3. It is important that a trustworthiness detector does not raise too many false positives (raising a suspicion event when no member is compromised). A false positive unnecessarily triggers a group membership protocol, which is typically quite complex and time-consuming. As a result, performance of a group communication system can degrade significantly if the trustworthiness detector reports too many false positives.
4. A compromised group member can try to defeat the trustworthiness detection process. In fact, an adversary may be able to attack and compromise multiple group members simultaneously. This is more likely in a group communication service in particular, because all group members typically maintain similar data structures and run similar protocols. If multiple group members are compromised with in a short time interval, a coordinated attack from multiple group members to defeat the trustworthiness detection process is possible.

To address these challenges, we have designed our trustworthiness detector based on the two important principles: focus on the observable effects, and detection in depth.

3.1 Focus on Observable Effects

The most important challenge in building a trustworthiness detector is that it is not clear what exactly trustworthiness of a computing component means. What observations can lead a trustworthiness detector to conclude that a group member can no longer be trusted? There are several components of trustworthiness that we understand at present. For example, any type of component failure, e.g. a crash failure [6], clearly indicates that such a component cannot be trusted to provide the expected functionalities. However, can we trust a component that is under denial of service attack?, or can we trust a component that *may have* sent a garbled message?

To address this, we observe that the design of a fault-tolerant system is typically based on the observable behavior of a component (failure models). It is important to note that failure models do not focus on causes of failures. For example, failures may occur because of a disruption in the power supply, or a circuit malfunction in a component. Fault-tolerant systems do not attempt to detect these causes of component failures to tolerate failures. On similar lines, we have focused on the observable effects of a component in designing our trustworthiness detector. For example, an observable effect in a group communication system is whether a particular group member is

consistently delaying the stability of multicast messages. Another example of an observable effect in a group communication system is if a group member is significantly deviating from the multicast pattern [16] of the supported application.

In the proposed trustworthiness detector, we have incorporated a list of such observable effects that can arise in a group communication system. These effects are then correlated with other observable effects to suspect a group member.

3.2 Detection in Depth

A common technique to build secure systems is "defense in depth" that structures different security mechanisms in different levels. The basic idea is that an adversary will have to break multiple levels of security mechanisms to break into a system. A system structured using defense in depth is generally more difficult to break than if all those security mechanisms were put in a single module.

We have adopted a similar approach in designing our trustworthiness detector. The basic idea of *detection in depth* technique is to structure various mechanisms for detecting any signs of untrustworthiness in a component in several distinct levels. An intruder that attempts to defeat the trustworthiness detector will have to defeat the detection process built into each of these levels, which will be a very difficult task. In particular, the architecture of our trustworthiness detector consists of four distinct levels: (1) operating system level, (2) network level, (3) application level and (4) peer level.

At the operating system level, an appropriate OS-based intrusion detector is used, while at the network level, an appropriate network-based intrusion detector is used. Detection mechanism at the application level watches for signs of abnormal conditions such as one or more function invocations being blocked indefinitely, excessive number of timeouts, object crashes, etc. Finally, a peer-level detection mechanism consists of a set of several peer-level detectors, each running at a different node. This mechanism exploits the communication and communication pattern of group communication mechanism to suspect a group member.

3.3 Architecture

The overall architecture of our trustworthiness detector is shown in Figure 1. The central component of this architecture is a *Resolver*. An instance of a resolver runs on each node hosting a group member. It acts on outputs from four different detectors--- network level, OS level, application level, and peer level detectors. Based on a trustworthiness detection policy (provided as another input), the resolver cross-examines these outputs and determines if a suspicion event should be raised. This cross-examination allows trustworthiness detector to correlate the outputs from different detectors, and make an informed decision. In particular, this aids in reducing false positive rate.

Trustworthiness detection policy dictates how the resolver uses outputs from various detectors to determine if a suspicion event should be raised. In particular, separate weights are assigned to the outputs from each detector, and the resolver raises a suspicion event whenever the sum of these weights exceeds a threshold (assigned by the policy). In the current implementation, weight assigned to the output from the peer level detector is highest, and in fact it exceeds the threshold. This means that any suspicion reported by the peer-level detector results in a suspicion event being raised by the resolver. Weights assigned to the outputs from the other detectors are less than the threshold. These values are such that the sum of the weights assigned to any two detectors exceeds the threshold. In addition to the weights, a trustworthiness detection policy also includes timing information. This determines the length of time interval for which a suspicion reported by one of the detector remains in effect. Trustworthiness detection policy is designed to be dynamic in nature. This policy can be modified by security administrators to reflect the current security requirements of the application being supported.

The four levels of detectors are independent of one another. Each may be designed and implemented in a unique way. These detectors may run on the same node as the resolver, or on a different node. In our current implementation, we rely on the availability of detectors at the network, OS, and application levels. For example, Emerald [17] and Snort [22] provide intrusion detection mechanism at the network level, while eXpert-BSM [23] provides intrusion detection mechanism at the OS level.

A study of various intrusion detection systems shows that there is no standard format for the output of these systems. To address this, we have introduced an *Output Format* component in our architecture. This component translates and formats the output from a specific detector to a format that the resolver can understand. The design of this component naturally depends on the nature of the specific detector it is interfaced with, and so, a separate implementation of this component is needed for each detector.

We have designed and implemented a peer level detector, which exploits common communication and computation patterns of a group communication system. In the rest of this paper, we focus on its design, implementation, and performance evaluation.

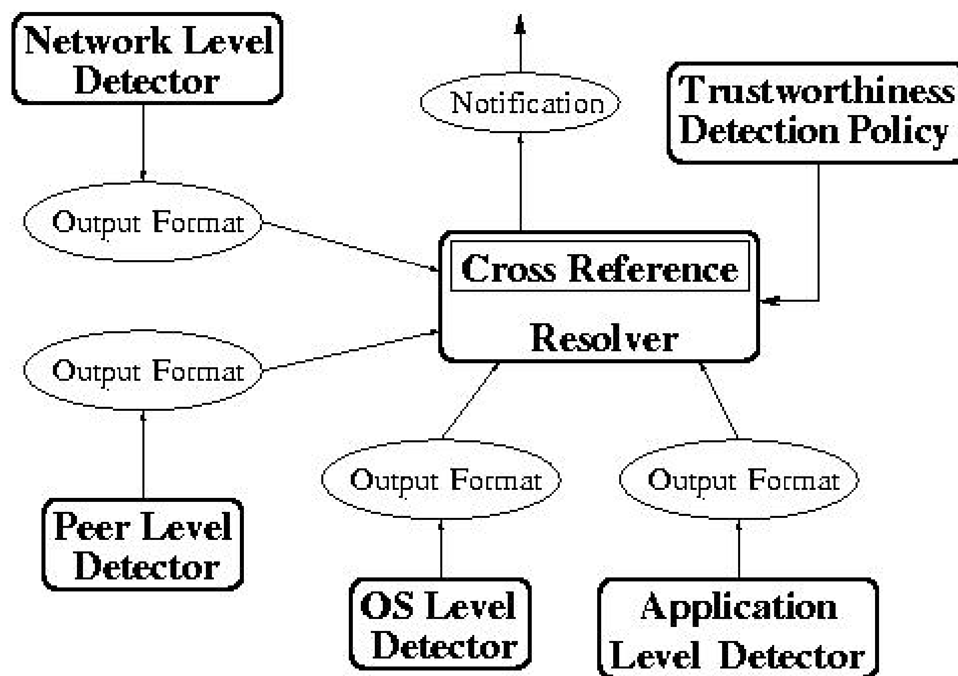

Figure 1: Architecture of the Trustworthiness Detector.

4 Peer Level Detector

Peer-level detection mechanism consists of a set of peer level detectors, each running at a node that hosts a group member. A peer level detector monitors one or more group members, and watches for signs of security compromise in those group members. An important property of a peer-level detector is that it runs on a node that is different from the nodes on which the group member(s) it is monitoring is (are) running. This provides some guard from an adversary that succeeds in compromising a node and attempts to defeat the detection mechanisms running on the same node.

In particular, a peer level detector watches for conditions that seem abnormal in a group communication environment. Figure 2 illustrates a high-level architecture of a peer level detector. The central component of this detector is a *Peer Level Resolver*. A peer level resolver receives inputs from several sources. These include a *Failure Detector* running on the same node, broadcast protocol of the supported group communication system,

peer level detectors running at other nodes, and a *Peer Level Detection Policy*. Based on these inputs, a peer level detector determines if any of the observable effects (See Section 3.1) that can be used to suspect a group member is present.

A failure detector watches for the failure of other group members. The types of failures this component typically detects include crash and performance failures [6]. Design and implementation of a failure detector for a group communication system has been extensively studied. For example, see [3,4]. In current group communication systems, a failure detector is typically implemented by a heartbeat protocol, wherein each group member periodically sends an *I am alive* message to either all other group members, or its neighbors in a (logical) cyclic order.

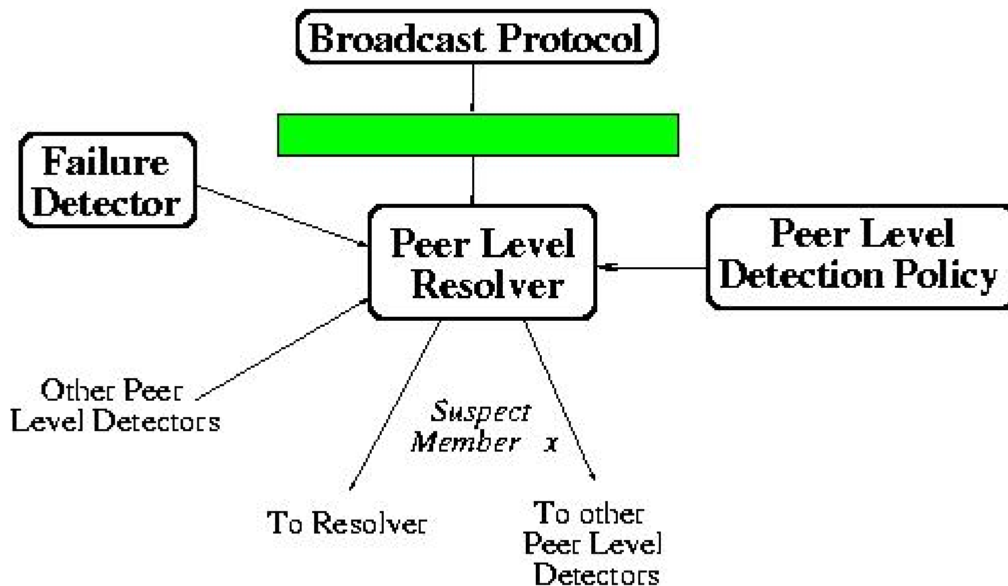

Figure 2: Architecture of the Peer Level Detector.

An important feature in the design of our peer level detector is that there is clear logical separation between the broadcast protocol and the detector. In particular, a broadcast protocol only provides certain relevant state information to the detector, and it is up to the detector to analyze this information, perhaps by correlating it with information from other sources to determine if a particular group member should be suspected. Communication from the broadcast protocol to the detector is accomplished via a shared data structure (*state communicator*) between the peer level resolver and the broadcast protocol. State communicator keeps record of incoming and outgoing messages. The broadcast protocol updates state communicator with the relevant state information, and the peer level resolver pulls this information to determine if any of the observable effects is present. The information stored in the state communicator naturally depends on the nature of the broadcast protocol and the failure scenarios being detected. Section 5 provides examples of such information and relates it to the observable effects for three different broadcast protocols: positive acknowledgement, negative acknowledgement, and token based broadcast protocols.

The peer level resolver is periodically triggered to check the information in the state communicator. It makes appropriate decisions based on a *peer level detection policy* that is provided as an input. The goal of having a peer level detection policy is to adapt the decision process of the peer level detector to suit different applications and computing scenarios. In the current implementation, this policy defines various thresholds and related data.

Output from a peer level detector is directed to the resolver of the trustworthiness detector that is running on the same node (see Figure 1), and the peer level detectors running on other nodes. This output is of the form

<*suspect member x; a threshold value*>. The threshold value is calculated by using the information included in the peer level detection policy.

5 Prototype Implementation

To test the applicability and effectiveness of the peer level detector, we have interfaced it with three different atomic broadcast protocols: a positive acknowledgement (PA), a negative acknowledgement (NA), and a logical ring-based (Ring) protocol. In this section, we give a brief overview of these three protocols, provide a list of observable effects in a group communication system that the peer level detector is looking for, and discuss specific mechanisms used to detect these effects in the three protocols.

5.1 Protocol Overview

PA

A broadcast is initiated by a group member (sender) by sending a broadcast message to all group members. This message includes a local sequence number that uniquely identifies the message sent from this particular sender. On receiving a broadcast message, each group member sends a positive ack to the sender to acknowledge the receipt of this message. The sender starts a timer after sending the broadcast message. When the timer expires, the sender checks if it has received positive acks from all group members. It resends the broadcast message to all those group members from whom a positive ack has not been received. This process continues until a positive ack has been received from all group members.

One specific group member in the group is designated as a *sequencer*. On receiving a broadcast message, the sequencer sends an *ordering* message to all group members. This message associates a global sequence number to the broadcast message. Group member deliver broadcast messages in the order of their associate global sequence numbers. Once again, group members acknowledge the receipt of ordering message by sending a positive ack to the sequencer, and the sequencer keeps resending the ordering message at periodic intervals to all those group members from whom a positive ack has not been received.

NA

As in PA, a broadcast is initiated by a group member (sender) by sending a broadcast message to all group members. This message includes a local sequence number that uniquely identifies the message sent from this particular sender. Also a designated sequencer sends the ordering message as in PA. However, the responsibility to detect missing messages belongs to the receiver in NA. If the receiver detects a gap in sequence number of incoming messages, it sends a retransmission request for the missing message to the sender. On receiving a retransmission request, a sender resends the requested message. Sender assumes that the receiver has received all messages if there is no request for retransmission.

Ring

In Ring, all group members are organized to form a logical ring. A special message, called *Token*, is circulated among group members. Only a group member that holds the token can send a broadcast message that contains both the local and global sequence numbers. In general, a sender has to buffer all outgoing messages until it holds the token. Each member holds the token for a specified time and forwards it to the next member in the logical ring. We use negative acknowledgement technique to provide recovery from lost messages. If a receiver detects a gap in sequence numbers of incoming messages, it buffers a retransmission request. Once that receiver holds the token, all pending messages including retransmission requests and replies to retransmission requests are sent.

5.2 Observable Effects

A compromised group member can launch four different types of attacks to either corrupt the state of a group communication system, or make the system unavailable to its clients. These attacks are are:

1) A compromised group member can launch a denial of service attack by attempting to saturate a group communication system by sending a large number of either legitimate group communication messages, or illegitimate (junk) messages at a very fast rate.
2) A compromised group member can launch a denial of service attack by attempting to block the progress of a group communication system by holding back some key messages, i.e. by not sending some messages it is expected to send.
3) A compromised group member can attempt to corrupt the state of a group communication system by sending legitimate (but incorrect) group communication messages.
4) A compromised group member can send incorrect information to other group members via various trustworthiness detector components that are running on the same node. These components include failure detector, peer level detector, and trustworthiness resolver.

The peer level resolver or the trustworthiness resolver is expected to detect any failure scenarios that arise out of the fourth type of attack, i.e. attacks via various trustworthiness detector components. Recall that a peer level detector and a trustworthiness resolver correlate the information they receive from various sources to decide if a suspicion event should be raised. In particular, a peer level resolver can assign a low weight to the inputs received from other peer level detectors to vote out an incorrect input received from a compromised group member.

Tables 1, 2, and 3 show the possible attack methods that a compromised group member may employ to saturate a group communication system by sending legitimate group communication messages or junk messages, block the progress of a group communication system, and corrupt the state of a group communication system respectively. These tables also show the corresponding observable effects, and the information communicated by respective atomic broadcast protocols to the peer level resolver via the state communicator. The compromised member is assumed to be member *y* here, and the peer level detector attempting to detect security compromises is assumed to be running on member *x*. The peer level detector employs several constants. These include message counts *M1*, *M2*, *M3*, *M4*, *M5*, *M6*, *M7*, *M8*, *M9*, *M10*, and *M11*, and timer values *T1*, *T2*, *T3*, *T4*, *T5*, *T6*, and *T7*. Values of these constants are provided by security administrators via the peer level detection policy file.

A compromised group member *y* can attempt to saturate the group communication state by sending either legitimate messages or junk messages at a very fast rate. Attacks using legitimate group communication messages include (1) repeatedly broadcasting the same message, (2) repeatedly broadcasting the same ordering message, if *y* is the sequencer, (3) repeatedly sending positive acks for the same broadcast message, (4) repeatedly sending retransmission requests for the same broadcast/ordering message, and (5) repeatedly broadcasting new (fabricated) messages. The observable effect that result from all these attacks is that the number of times a group member receives a particular message or a particular type of message exceeds a certain a priori fixed limit. To enable the peer-level detector make recognize this, the broadcast protocol communicates the appropriate message count via the state communicator.

A compromised group member *y* can attempt to block the progress of a group communication system by (1) not retransmitting a message after receiving a retransmission request, (2) not sending a positive ack after receiving a broadcast or ordering message, (3) not sending an ordering message, (4) not forwarding the token, and (5) not broadcasting a message after receiving an update from its client. Table 2 shows the corresponding observable effects, and the information communicated by respective atomic broadcast protocols to the peer level resolver via state communicator. The observable effects that result from all these attacks are that the group member does not receive a particular message or a particular type of message within a certain a priori fixed time limit. To enable the peer-level detector make recognize this, the broadcast protocol communicates the appropriate timeout information to the state communicator.

Attack Methods	Observable Effects (at member x)	Information communicated (member x)		
		PA	NA	Ring
Repeatedly broadcast the same message.	x receives the same broadcast message more than *M1* times.	Number of times x receives the same broadcast message sent by y, even after repeatedly sending an acknowledgement.	Number of times x receives the same broadcast message sent by y, even when x doesn't send any retransmission request.	Number of times x receives the same broadcast message sent by y, even when x doesn't send any retransmission request.
Repeatedly send the same ordering message.	x receives the same ordering message more than *M2* times.	Number of times x receives the same ordering message sent by y.	Number of times x receives the same ordering message sent by y.	N/A
Repeatedly send the same positive ack.	x receives positive acks for the same broadcast message more than *M3* times.	Number of times x receives a positive ack from y for the same broadcast message or ordering message broadcast by x.	N/A	N/A
Repeatedly send retransmission requests for messages already received.	x receives retransmission requests for the same broadcast message or ordering message more than *M4* times.	N/A	Number of times x receives retransmission requests from y for a broadcast message or ordering message broadcast earlier by x.	Number of times x receives retransmission requests from y for a message broadcast earlier by x.
Repeatedly broadcast the new (fabricated) messages.	x receives more than *M5* new broadcast messages from y in the last *T1* time units.	Number of new broadcast messages x has received in the last *T1* time units.	Number of new broadcast messages x has received in the last *T1* time units.	Number of new broadcast messages x has received in the last *T1* time units.
Repeatedly broadcast junk messages.	x receives more than *M6* junk messages from y in the last *T2* time units.	Number of junk messages x has received in the last *T2* time units.	Number of junk messages x has received in the last *T2* time units.	Number of junk messages x has received in the last *T2* time units.

Table 1. Denial of service attack by attempting to saturate the group communication system.

Attack Methods	**Observable Effects (at member *x*)**	**Information communicated (member *x*)**		
		PA	**NA**	**Ring**
y does not retransmit a message after receiving a retransmission request.	*x* does not receive a message detected to be missing even after sending *M6* retransmission requests.	N/A	Number of times *x* sends retransmission requests to *y* to request a missing message	Number of times *x* sends retransmission requests to *y* to request a missing message
y does not send a positive ack after receiving a broadcast or ordering message.	*x* times out *M7* times after sending a broadcast or ordering message	Number of times *x* times out after sending a broadcast or ordering message and not receiving a positive ack from *y*.	N/A	N/A
y does not send an ordering message	*x* does not receive an ordering message for a broadcast message it received more than *T2* time units earlier.	After receiving a broadcast message from some member, *x* does not receive an ordering message from (sequencer) *y* even after *T2* time units have elapsed.	After receiving a message from some member, *x* does not receive an ordering message from (sequencer) *y* even after *T2* time units have elapsed.	N/A
y does not forward token.	*x* does not receive a token transfer message	N/A	N/A	No token transfer takes place after *T3* time units have elapsed after *y* became the token holder.
y does not broadcast a message.	*X* receives less than *M8* new broadcast messages from *y* in the last *T4* time units.	Number of new broadcast messages received from *y* in the last *T4* time units.	Number of new broadcast messages received from *y* in the last *T4* time units.	Number of new broadcast messages received from *y* in the last *T4* time units.

Table 2. Denial of service attack by attempting to block the progress of group communication system.

Finally, a compromised group member *y* can attempt to corrupt the state of a group communication system by (1) sending incorrect broadcast messages, *e.g.* attaching out-of-order local sequence number, or an already used local sequence number, (2) sending incorrect ordering message, *e.g.* by attaching an out-of-order global sequence number, or an already used global sequence number, and (3) sending the same broadcast or ordering message to different members, but attaching a different local/global sequence number in the copies sent to different members.

An important issue in detecting if a group member is attempting to corrupt the state of the group communication system is how any of these attacks is detected. In our design of trustworthiness detector, we leave this to the atomic broadcast protocol. For example, the atomic broadcast protocol can detect some of these attacks by using appropriate cryptographic algorithms or security protocols. The philosophy behind the design of the peer-level detector is that the higher-level protocols provide appropriate information including any abnormal conditions, and the peer level detector makes a decision based on this information and the information it receives from other sources.

Attack Methods	**Observable Effects (at member *x*)**	**Information communicated (member *x*)**		
		PA	**NA**	**Ring**
Send incorrect broadcast messages, *e.g.* attaching out-of-order local sequence number, or an old local sequence number.	*x* received *M9* broadcast messages from *y* containing incorrect local sequence number in the last *T5* time units.	Number of broadcast messages *x* received from *y* containing incorrect local sequence number in the last *T5* time units.	Number of broadcast messages *x* received from *y* containing incorrect local sequence number in the last *T5* time units.	Number of broadcast messages *x* received from *y* containing incorrect local sequence number in the last *T5* time units.
Send incorrect ordering message, *e.g.* by attaching incorrect global sequence number	*x* received *M10* broadcast messages from *y* containing incorrect global sequence number in the last *T6* time units.	Number of ordering messages *x* received from *y* containing incorrect global sequence number in the last *T6* time units.	Number of ordering messages *x* received from *y* containing incorrect global sequence number in the last *T6* time units.	Number of ordering messages *x* received from token holder *y* containing incorrect global sequence number in the last *T6* time units.
Send the same broadcast or ordering message to different members, but attach a different local/global sequence number in the copies sent to different members.	*x* discovers *M11* times this attack from *y* in the last *T7* time units.	Number of times *x* discovers this attack from *y* in the last *T7* time units.	Number of times *x* discovers this attack from *y* in the last *T7* time units.	Number of times *x* discovers this attack from *y* in the last *T7* time units.

Table 3. Attempts to corrupt the state of the group communication system.

6 Implementation and Performance

We have implemented the peer-level detector and integrated it with three broadcast protocols as described in the last section using NS2. In this simulation, we used a static group membership, since we wanted to measure the overhead that occurs from the detection mechanism only. Group members ranged from 4 to 12. The topology used

is a mesh network, except for the token ring protocol. Each simulation experiment was run for 30 seconds, with a fault injected 10 seconds after the start. Simulated faults were injected into the system one by one, and the compromised group member(s) was (were) randomly selected. A compromised group member launched one of the following types of attack:

(a) DOS attack by saturating the group communication system
   i. It sends repeated broadcast message (PA, NA, Ring).
   ii. It sends repeated ordering message (PA, NA).
   iii. It sends repeated positive acks (PA).
   iv. It sends repeated retransmission requests (NA, Ring).
(b) DOS attack by blocking the progress of the group communication system
   i. It does not resend message after being requested (NA, Ring).
   ii. It does not resend message after time out (PA).
   iii. It does not send positive ack (PA).
   iv. It does not send ordering (PA, NA).
   v. It does not forward token (Ring).
(c) Corrupting the state of the group communication system
   i. It attaches different local sequence number to the same broadcast message when sending to different group members (PA, NA, Ring).
   ii. It attaches different global sequence number to the same ordering message when sending to different group members (PA, NA, Ring).
   iii. It attaches incorrect local sequence number in the broadcast message (PA, NA).
   iv. It attaches incorrect global sequence number in the ordering message (PA, NA, Ring).
(d) Combination of (a) and (b).
(e) Combination of (a) and (c).
(f) Combination of (b) and (c).
(g) Combination of (a), (b), and (c).

To evaluate the performance of our trustworthiness detector, we simulated compromise of a single member and two members (group size > 5). We have measured the following four performance indices:

1. The time it takes for one correct group member to suspect one compromised group member.
2. The time it takes for more than one-third of the group members to suspect one compromised group member.
3. The time it takes for one correct group member to suspect two compromised group members.
4. The time it takes for more than one-third of the group members to suspect two compromised group members.

In all our experiments, we used a very simple peer level detection policy, which is plugged in as another input file and is read into the program before simulation. For example, the peer level detection policy used in PA is as follows. Similar policies were used in NA and Ring.

```
#configuration parameter
set M1 10; maximum number of times the same broadcast message is received
set M2 10; maximum number of times the same ordering message is received
set M3 10  ; maximum number of positive acks for the same local sequence number
set M4  5; maximum number of times  an ORDER message is sent.
set M5 100; maximum number of new broadcast messages received from group member in last one second
set M6 2; maximum number of junk messages in last one second.
….
….
```

Performance of our trustworthiness detector under different computing environments is shown in Figures 3—11. All these figures report detection time to suspect one or more compromised group members. While the faults related to each attack method were injected one at a time, the figures show the average detection time under all attack methods with in a particular type, i.e. all attack methods that cause saturation are grouped as saturation attacks, and so on. We will analyze this performance by comparing appropriate graphs. First, we compare the graphs shown in Figures 3, 4, and 5. These graphs show the time it takes for a first member to suspect a member attempting to saturate a group communication system, block the progress of a group communication system, and corrupt the state of a group communication system respectively. This detection time is plotted as a function of group size. In all these experiments, a single group member was simulated to be malicious by launching one of the three types of attacks. The message arrival pattern in all these experiments was assumed to be *uniform*, i.e. all group members initiated message broadcasts at the same average rate.

There are several observations we make from these graphs. First, as expected, the detection time increases with increase in group size. This is attributed to the increase in number of message exchanges that take place as a result of larger number of group members. Also, the peer level resolver needs to correlate the observation about suspicion of a group member with other peer level detectors. When group size is large, time to do this correlation is large. Second, the detection time in Ring is significantly larger than the detection times in PA or NA. The main reason for this is that a group member must wait until it receives the token to send any protocol/control messages in Ring. This wait time is in addition to the time it takes to suspect a member compromise. Unlike Ring, there is no extra wait involved for a member to send any protocol messages in PA or NA. The overall effect of this extra wait time is that the time to suspect a compromised group member in Ring is significantly larger than in PA or NA.

The third observation is that while the detection times in PA and NA are similar, they are slightly larger in NA than in PA under a blocking attack, and slightly smaller in NA than PA under a saturation attack. This can be explained by looking at the number of messages exchanged in the two protocols. In the absence of any message losses, the number of message exchanges in PA is larger than the number of message exchanges in NA. In general, it is much easier to detect blocking attacks if there is a lot of activity in a protocol. This is because any attempt to block progress in a group communication system where protocol activity is high results in a markedly reduced activity, and the trustworthiness detector immediately notices it. Since, more messages are exchanged in PA, blocking attacks are detected faster in PA. On the other hand, it turns out that saturation attacks are easier to detect when the normal protocol activity is low. This is because any attempt to saturate in a group communication system where protocol activity is low results in a markedly increased activity, and the trustworthiness detector immediately notices it. Hence, saturation attacks are detected faster in NA.

The fourth observation is that in all protocols, it takes longer time to detect blocking attacks than saturation or state corruption attacks. This is a consequence of various timeout values set in the protocols and the trustworthiness detector. However, we believe that this is a general case. Typically, timeout values are set conservatively in a trustworthiness detector to reduce the number of false positives. This is particularly true for an asynchronous distributed computing system. Since a blocking attack is typically detected by using an appropriate timeout mechanism, it is highly likely that the detection time will be longer.

Finally, we observe that in all protocols, a state corruption attack is detected the fastest. The main reason for this is our reliance on the strength of cryptography. We have assumed in our system design that appropriate cryptographic techniques exist to detect any attempt to corrupt the state of a group communication system. These include sending junk messages, incorrect local/global sequence numbers, and different local/global sequence numbers. As a result, any attempt to corrupt the state of a group communication system is detected quite fast by our trustworthiness detector. We note that cryptographic techniques exist to detect all these types of attacks. For example, see [2,12,19,20]. Also, it should be noted that using cryptography has significant impact on the performance of a group communication system [19]. However, this cost is incurred in handling both legitimate and illegitimate messages. For this reason, we have not attributed this cost to the performance of overhead our trustworthiness detector.

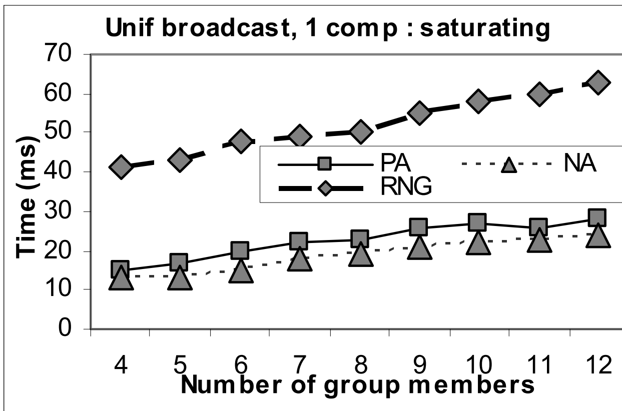

Figure 3: Time for a first member to suspect a single group member attempting to saturate the group communication system.

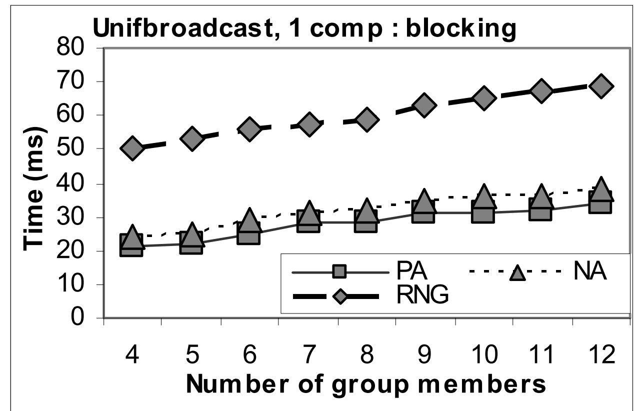

Figure 4: Time for a first member to suspect a single group member attempting to block a group communication system.

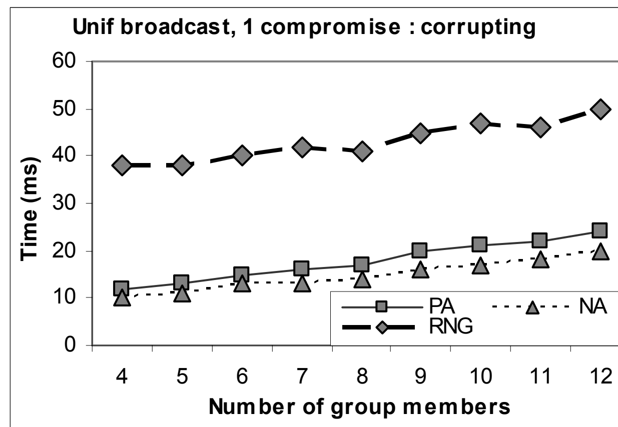

Figure 5: Time for a first member to suspect a single group member attempting to corrupt the state of a group communication system.

We now compare the graphs shown in Figures 3, 6, and 7. The graphs plot detection time as function of group size under that same type of attack (saturation attack) for three different message arrival patterns –*uniform*, *one-active*, and *bursty* message arrival patterns respectively. As mentioned earlier, under a uniform message arrival pattern, all group members initiate message broadcasts at the same average rate. Under a one-active message arrival pattern, one group member initiates all message broadcasts at a given (average) arrival rate. Finally, under a bursty message arrival pattern, all group members alternate between a short (bursty) period when they initiate message broadcasts at a fast rate and a relatively long (idle) period when they initiate message broadcasts at very slow (almost zero) rate. The bursty periods of different group members may or may not overlap (either partially or completely).

The main observation we make from these three graphs is that in all three protocols, the detection time is largest under a one-active message arrival pattern and smallest under a uniform message arrival pattern. The reason for this behavior is the following. If the normal rate of sending messages is low for a group member, it is easier (and faster) to detect any saturation attack launched by that member. Under one-active message arrival pattern, the compromised member's (the active member in our experiments) normal sending rate is quite high. As a result, it takes long time to detect any saturation attack from this member. On the other hand, under a uniform message

arrival pattern, the compromised member's normal sending rate is lower than under one-active or bursty message arrival patterns. As a result, it takes a relatively shorter time to detect any saturation attack under uniform message arrival pattern. Exactly, the opposite behavior was observed for detecting blocking attacks. Detection time was largest under uniform message arrival pattern and smallest under one-active arrival pattern. This once again confirms that it is easier to detect blocking attacks when there is low protocol activity.

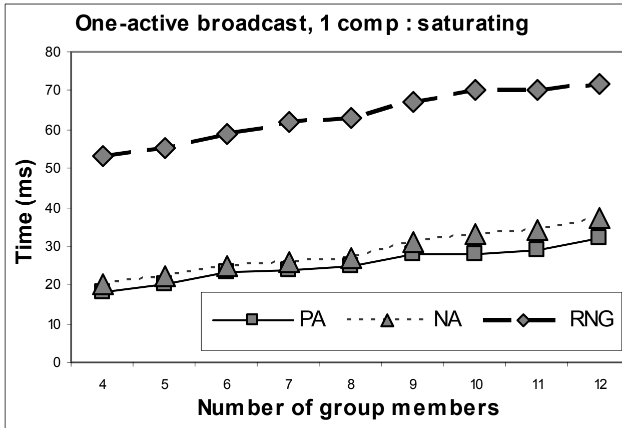

Figure 6: Time for a first member to suspect a single group member attempting to saturate a group communication system.

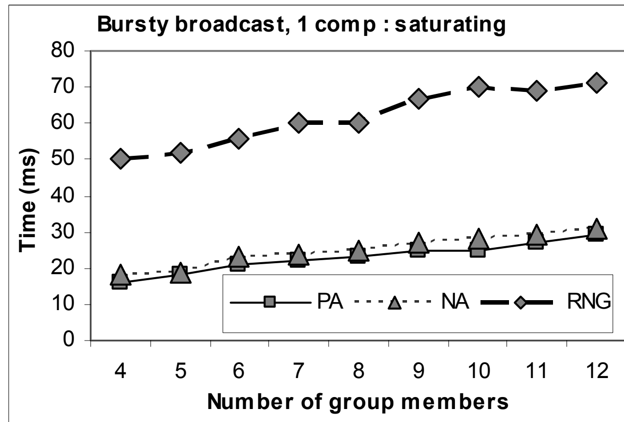

Figure 7: Time for a first member to suspect a single group member attempting to saturate a group communication system.

We now analyze performance shown in Figures 3 and 8. These graphs plot the detection time by a first member and 1/3rd of the group members under a uniform message arrival pattern and saturation attack by a single compromised group member. Detection time for suspecting a compromised group member by 1/3rd of the group members is important, because intrusion-tolerant group membership protocols initiate the creation of a new group only after more than 1/3rd of the group members have suspected a member, e.g. see [12, 19, 20]. The main reason for this is that it has been shown that in order to achieve consensus in the presence of Byzantine failures, no more that 1/3rd of the group members may fail [11].

We make two important observations from these graphs. First, the detection time to suspect a compromised group member by 1/3rd of the group members is significantly higher than the detection time by a first group member. For example, when there are 12 group members, the detection time by a first group member in PA or NA is around 20 ms, while it is more than 40 ms for 1/3rd of the group members. While it is expected that the detection time to suspect a compromised group member by 1/3rd of the group members will be higher than the detection time by a first group member, the amount of difference was a surprise to us. The second observation we make is that the detection time to suspect a compromised group member by 1/3rd of the group members increases significantly when the number of group members increases from 5 to 6, 8 to 9, or 11 to 12. The main reason is that the value of 1/3rd of group members increases when the group size reaches 6, 9, or 12. In fact, we expect this behavior for any group size that is a multiple of 3.

We now look at the performance of our trustworthiness detector when multiple group members are compromised simultaneously. Graphs shown in Figures 9, 10, and 11 plot detection times under uniform message arrival pattern and 2-member compromise. In these graphs, NA has been used and all combinations of two attack types have been considered. Figure 9 plots the detection time to suspect one compromised group member by a first group member when two members have been compromised. Figure 10 plots the detection time to suspect both compromised group members by a first group member. Finally, Figure 11 plots the detection time to suspect both compromised group members by 1/3rd of the group members when two members have been compromised.

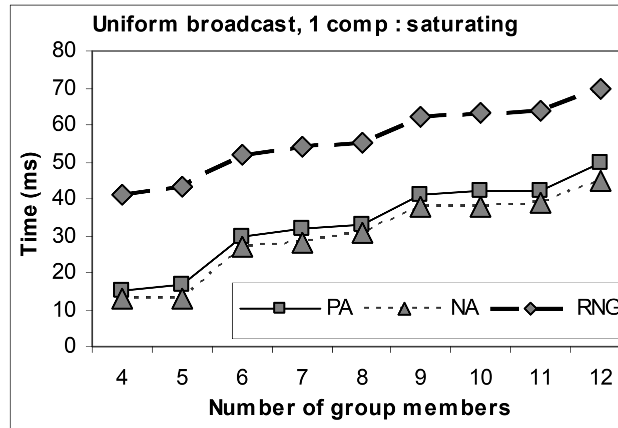

Figure 8: Time for 1/3rd of the group to suspect a single group member attempting to saturate a group communication state.

The first observation we make from these graphs is that some of the general observations we made under a single member compromise hold for multiple member compromise as well. These include increase in detection time with increase in group size, significantly larger time to suspect by 1/3rd of the group members than by a first group member, and sudden increase in detection time for group sizes that are multiples of 3. We also observe that it takes longer to suspect both compromised group members than one group member (as expected), and the difference in these to detection times is only moderate. The second observation we make from these graphs is that any attack combination involving blocking attack takes longer to detect. This once again shows that it is more difficult and time-consuming to detect blocking attacks than other types of attacks.

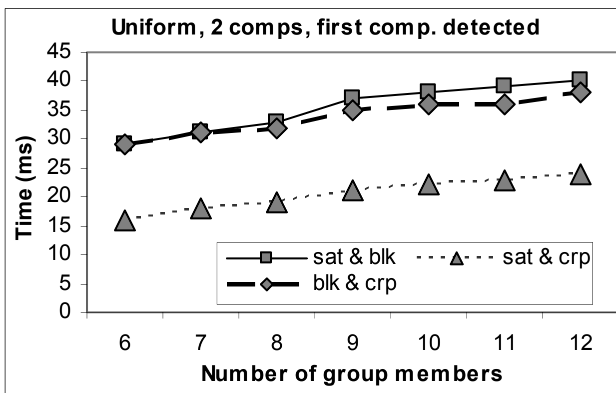

Figure 9: Time for a first group member to suspect one group member when two members have been compromised.

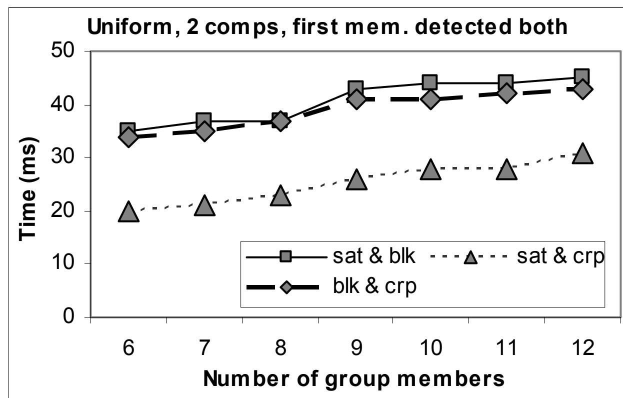

Figure 10: Time for a first group member to suspect both group members when two members have been compromised

Finally, by comparing the graphs shown in Figure 3, 4, and 5 with that graph shown in Figure 9, we notice that a combination of attacks generally take more time to be detected than a single attack type. The difference is more pronounced when the combination involves blocking attack.

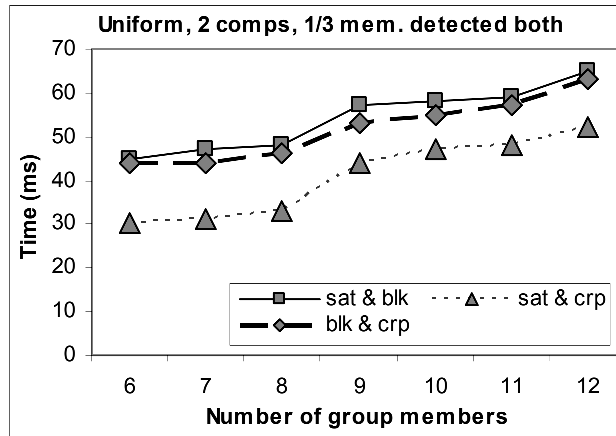

Figure 11: Time for 1/3rd of the group to suspect both group members when two members have been compromised.

7 Discussion

In this paper, we have described the design, implementation, and performance evaluation of a trustworthiness detector for an intrusion-tolerant group communication service. This detector is generic in the sense that it is independent of the actual broadcast or group membership protocol being supported by the group communication system. The system design provides a clean separation between the detector and the atomic broadcast protocol, and facilitates communication between the two components via a shared data structure called state communicator. We have demonstrated the flexibility and usefulness of this detector by incorporating it in three different group communication systems. The trustworthiness detector proposed in this paper is a part of a larger project aimed to design an intrusion-tolerant middleware service based on group communication.

The paper provides an extensive performance evaluation of the proposed trustworthiness detector. While the actual detection time values reported in the paper are not so important, as they naturally depend on many factors such as timer values, their relative behavior under different operating conditions and attack types is very useful. An important contribution of this paper is the extensive performance evaluation of a stand-alone trustworthiness detector specifically designed for a group communication, and the insight it has provided in detecting malicious behaviors.

The first conclusion from our performance evaluation is that there is a close relation between the amount of protocol activity in terms of number of message exchanges and the time it takes to detect specific types of attacks. If there is a large amount of protocol activity, e.g. by either the message arrival rate or pattern, or the control messages exchanged in the protocol, it is more difficult and time consuming to detect saturation attacks and easier to detect blocking attacks. To this end, it pays to have some semantic information about the application being supported in the trustworthiness detector. For example, what kind of message arrival pattern will be exhibited by an application? The trustworthiness detector provides a mechanism to input such semantic information via a configurable policy file.

The second conclusion is that it is generally more difficult and time consuming to detect blocking attacks than saturation or state corruption attacks. This indicates that a trustworthiness detector should pay close attention to the quieter members in a group. The third conclusion is that it takes a relatively shorter period of time to detect a state corruption attack than blocking or saturation attacks, if appropriate cryptographic techniques have been used in the broadcast protocol.

Our final conclusion is that the detection time to suspect a compromised group member by 1/3rd of the group members is significantly larger than the detection time by a first group member. This has a very important consequence in the design of an intrusion-tolerant group membership protocol. To ensure that a malicious group member is not raising an incorrect suspicion event, a group membership protocol should wait until more than 1/3rd of the group members have suspected a group member before attempting to form a new group. Otherwise, a compromised group member can launch a denial of service attack by simply (and falsely) suspecting another group member. Because of this reason, all intrusion-tolerant group membership protocols that we know wait until more than 1/3rd of the group members have suspected a group member. However, doing so allows a compromised group member to attack a group for a longer period of time. As shown here, there is a significant time gap between one member suspecting a member and 1/3rd of the group members suspecting the same member. An interesting question is can we design an intrusion-tolerant group membership protocol that does not wait until 1/3rd of the group members have suspected a member, but still avoids a possibility of the above-mentioned denial of service attack? We are currently addressing this issue in the design of our intrusion-tolerant middleware service.

References

[1] L. Alvisi, et al. Fault Detection for Byzantine Quorum Systems. *IEEE Transaction on Parallel and Distributed System*, 12:996-1007, 2001.

[2] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In the *Proceedings of OSDI*, 1999.

[3] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, pages 147-158, Aug 1992.

[4] T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325-340, Aug 1991.

[5] T. Courtney, et al, M. Atighetchi, et al, M. Cukier, and J. Gossett. Providing intrusion tolerance with ITUA. In *Proceedings of the 1st Workshop on Intrusion Tolerant Systems*, Washington D.C., June 2002.

[6] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2): 56-78, Feb 1991.

[7] M. Cukier, et al. Intrusion Tolerance Approaches in ITUA. In *Supplemental of the 2001 International Conference on Dependable Systems and Networks*, pages B64-B65, 2001.

[8] A. Doudou, et al. Muteness Failure Detectors: specification and implementation. In *Proceeding of the 3rd European Dependable Computing Conference*, Sep 1999.

[9] B. Dutertre and V. Crettaz. Intrusion-tolerant enclaves. In *The 2002 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[10] B. Dutertre, H. Saidi, and V. Stavridou. Intrusion-tolerant group management in enclaves. In *International Conference on Dependable Systems and Networks (DSN'01)*, pages 203-212, Goteborg, Sweden, July 2001.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. In *Journal of ACM*, 35(2):288-323, Apr 1988.

[12] K. Kihlstrom, L. Moser, and M. Melliar-Smith. The securing protocols for secure group communication. In *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences*, Kona, Hawaii, Jan 1998.

[13] K. Kihlstrom, L. Moser, and M. Melliar-Smith. Byzantine Fault Detectors for Solving Consensus. In *The Computer Journal*, 46(1), 2003.

[14] D. Malkhi, and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceeding of the 10th Computer Security Foundations Workshop*, Rockport MA, June 1997.

[15] S. Mishra. A middleware for constructing highly available, fault-tolerant, and attack tolerant services. In *Proceedings of the 17th ISCA International Conference on Computers and Applications*, San Francisco, CA, April 2002.

[16] S. Mishra and L.Wu. An evaluation of flow control in group communication. *IEEE/ACM Transaction on Networking*, 6(5), Oct 1998.

[17] P. A. Porras, and P. G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 19th National Information Systems Security Conference*, Baltimore, MD, October 1997.

[18] R. Power. 2002 CSI/FBI computer crime and security survey. Technical report, Computer Security Institute Publication, Spring 2002. Available at http://www.gocsi.com/forms/fbi/pdf.html.

[19] H. Ramasamy, P.Pandey, J. Lyons, M. Cukier, and W. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. In *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks,* Washington D.C., June 2002.

[20] M. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems,* volume LNCS 938. Springer Verlag 1995.

[21] P. Verissimo, N. Neves, and M. Correia, The Middleware Architecture of MAFTIA: A Blueprint. In *Proceedings of the IEEE Third Information Survivability Workshop*, Boston, MA, Oct 2000.

[22] Snort. The Open Source Intrusion Detection System. Available at *http://www.snort.org*.

[23] eXpert-BSM- Intrusion detection system for Solaris. Available at *http://www.sdl.sri.com/projects/emerald/releases/eXpert-BSM/eXpert-BSM-DataSheet.pdf*