The Byzantine Agreement Protocol Applied to Security

by

David M. Toth

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

December 2004

APPROVED:

____

Professor Fernando C. Colón Osorio, Thesis Advisor

____

Professor Stanley M. Selkow, Thesis Reader

____

Professor Michael A. Gennert, Head of Department

Abstract

Intrusion Detection & Countermeasure Systems (IDCS) and architectures commonly used in commercial, as well as research environments, suffer from a number of problems that limit their effectiveness. The most common shortcoming of current IDCSs is their inability to tolerate failures. These failures can occur naturally, such as hardware or software failures, or can be the result of attackers attempting to compromise the IDCS itself. Currently, the WPI System Security Laboratory at Worcester Polytechnic Institute is developing a Secure Architecture and Fault-Resilient Engine (*S.A.F.E.*), a system capable of tolerating failures.

This system makes use of solutions to the Byzantine General's Problem, developed earlier by Lamport and others. Byzantine Agreement Protocols will be used to achieve consensus about which nodes have been compromised or failed, with a series of synchronized, secure rounds of message exchanges. Once a consensus has been reached, the offending nodes can be isolated and countermeasure actions can be initiated by the system.

We consider the necessary and sufficient conditions for the application of Byzantine Agreement Protocols to the intrusion detection problem. Further, a first implementation of this algorithm will be embedded in the Distributed Trust Manager (DTM) module of *S.A.F.E.* The DTM is the key module responsible for assuring trust amongst the members of the system. Finally, we will evaluate the DTM, as a standalone unit, to ensure that it performs correctly.

Acknowledgements

I would like to thank the people who have made this thesis possible. I would like to thank my advisor, Professor Fernando C. Colón Osorio, for piquing my curiosity about his project, *S.A.F.E.*, and for giving me a chance to work on it. I have learned a great deal from him about both my thesis topic and academia. Thanks to Professor Stan Selkow for volunteering to be the reader. I would like to thank Frank Posluszny, Mike Voorhis, Jesse Banning, Chuck Wilcox, Randy Robinson, and King Lam for their technical assistance and for helping me get a hold of the significant amount of hardware I needed. Thanks are also in order to Chris O'Connor and Tom Collins, who gave me the opportunity and ability to do some important testing on modern hardware in the ADP lab. I would like to thank my friends, family, and professors for their support. Most importantly, I want to thank my wife, Laura, without whose support, understanding, and sacrifices, I could not have ever done this thesis.

Contents

1 Introduction

1.1 Definition of Intrusion Detection

Intrusion detection is the ability to detect security breaches. Intrusion detection systems (IDSs) are useful in detecting successful breaches of the security systems present on a computer system, monitoring attempts to breach said security, and collecting information to allow the system to launch successful countermeasures. A successful countermeasure, within this context, is defined a set of actions taken by the system under attack to prevent the system from being compromised. We define a compromised system in this work as the opposite of a secure computer system, which is defined by Garfinkel and Spafford as "a system that can be depended upon to behave as it is expected to" [GS 91].

A more formal definition of Intrusion Detection is "the problem of identifying individuals who are using, or attempting to use a computer system without authorization (i.e., crackers) and those who have legitimate access to the system but are abusing their privileges (i.e., the insider threat)" [BI 94, BA 98]. In the last decade, two models of Intrusion Detection have been widely accepted [DE 87]. These are anomaly detection and misuse detection. Anomaly detection consists of monitoring system behavioral statistics and flagging behavior that is statistically abnormal with respect to normal system operation. Some examples of anomalies are a sudden spike in CPU usage or a login at a time when the system is normally idle. Misuse detection attempts to discover attacks that exploit weaknesses in the security of a system. These weaknesses include buffer overflow attacks, holes in default system configurations such as unnecessarily open ports, and design flaws such as those in the TCP/IP protocol.

1.2 Criteria for Robust Intrusion Detection Systems

Amongst the most important considerations and limitations present in the design of all such systems are the following set of problems.

1. **Feature Selection And Pattern Categorization**

   In both anomaly and misuse Intrusion Detection Systems, the optimal set of system parameters to be monitored (anomaly detectors) or design rules (signatures) to be matched (misuse detectors) are often assumed, a priori. Optimal is defined as maximum discrimination between intrusions and normal system behavior while minimizing the set of system parameters monitored.

2. **Adaptation**

   Systems have been built and deployed that deal very effectively with threats or intrusions previously reported or categorized. However, when previously unseen threats appear, the systems perform poorly. For example, in the 1999 DARPA - Off-Line Intrusion Detection Evaluation, it was reported that the systems under test failed to detect an attack in 17.2 % of the cases due primarily to lack of in-depth analysis of protocols, or because signatures of old attacks did not generalize to new attacks [LI 99, HA 01].

3. **Fault Tolerance And Resistance To Subversion**

   Computers fail due to accidental or malicious activities. The system being designed must be able to recover from the traditional forms of failures such as crashes and software failures, as well as being able to protect itself from deliberate attempts to compromise it.

4. **Performance**

   The system must impose minimal overhead on the computers it is protecting while running, and must be capable to sustain its performance characteristics under increasing loads and changes in the patterns of usage.

Each of the problems mentioned needs to be addressed in order to create a robust IDCS. It is clear from previous research and the evaluations performed, that in general, intrusion detection systems built on a distributed architecture, perform better than centralized systems [HA 01]. Spafford summarizes the advantages and disadvantages of distributed versus centralized systems [SP 00]. In this research, we have built a prototype of an IDCS based on a distributed architecture.

1.3 Current State of Intrusion Detection Systems (IDSs)

There are currently many commercial and open source IDSs and IDCSs available, such as Snort, Tripwire and Bro. It is clearly well beyond the scope of this work to present a survey of them, although such documents exist [TA 04, BU 02, AX 00, AC 99]. Instead, we give a brief description of how some of the well-known IDSs and IDCSs work.

Snort monitors network traffic and alerts the user whenever there is traffic that matches one of the rules the user has configured *Snort* to use [KO 03]. To render *Snort* inoperable and thus useless, an attacker must simply compromise Snort, because there is nothing protecting *Snort*. This can be done either by damaging or disabling the program, or with a denial of service (DOS) attack, by simply flooding the host *Snort* is protecting with packets.

Tripwire, which has both commercial and open source versions available, monitors file attributes that are expected to stay constant, such as the size and signature of binary files [TW 04]. While *Snort* runs continuously once it has been started, *Tripwire* does not. *Tripwire* must be run periodically, by running it manually or setting up a cron job or equivalent script to run it periodically. *Tripwire* is resource intensive enough that it cannot just be run every few minutes, so it is not constantly monitoring the system.

Thus, a computer that *Tripwire* is protecting can be compromised, and this will remain undetected by *Tripwire* until it is run again. Like *Snort*, because nothing is protecting *Tripwire*, it can be specifically targeted and compromised.

Bro is a stand-alone UNIX-based system that attempts to detect intrusions by monitoring network traffic [PA 99, BI 04]. It observes all the network packets and, like *Snort*, it uses rules that the administrator creates to detect intrusions [BI 04]. Because it is watching the network traffic, it can be crippled by flooding the network with traffic, which renders it unable to examine all the traffic [PA 99]. *Bro* is also vulnerable to attacks directed against the system where it is running, and furthermore, it is vulnerable to two computers networked together and working together to try to compromise it [PA 99].

The state of the art IDSs and IDCSs are clearly inadequate, as the flurry of both commercial and academic activity continues in the area. All the available systems we are aware of suffer from shortcomings similar to those in *Snort*, *Tripwire*, and *Bro*. The systems do not protect themselves from attack, do not learn how to prevent new attacks that are devised, and consume so much of a computer's resources that they prevent work from being done. Due to these shortcomings, we are attempting to create a framework for an IDCS that addresses the shortcomings of the currently available systems.

1.4 Related Work

The WPI System Security Research Lab is currently developing an innovative IDCS called *S.A.F.E.* The purpose of *S.A.F.E.* is to use proven techniques from the areas of distributed processing, computer architecture, and clustering algorithms, to build a framework for a viable, real-time IDCS that can perform in large networks of computers without significantly affecting the performance of the computers it is protecting. *S.A.F.E.* attempts to remedy the main problems with current IDSs. When completed, *S.A.F.E.* will be able to learn and adapt, thereby detecting new types of attacks that have not been identified and publicized. It will be fault tolerant and able to withstand attacks directly against itself, to protect itself from being compromised or disabled. *S.A.F.E.* has been designed to work on computers capable of running the Java 2 Standard Edition (J2SE) runtime environment from Sun Microsystems. This allows *S.A.F.E.* to run on computers with the Windows, Linux, Solaris, and MacOS operating systems, as well as any others that have J2SE runtime environments from Sun Microsystems. We provide a brief overview of the *S.A.F.E.* system and one of its major components, the Secure Host Manager (SHM), but our focus is on the portion of *S.A.F.E.* that needs to be completed - the Distributed Trust Manager (DTM). The architecture of *S.A.F.E.* is shown in figure 1.

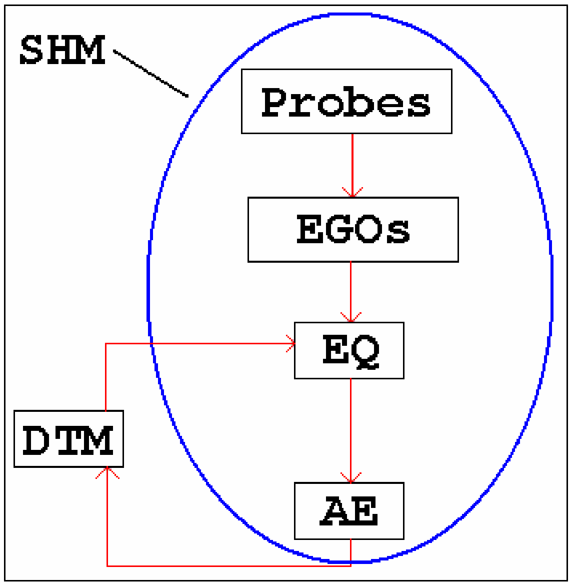
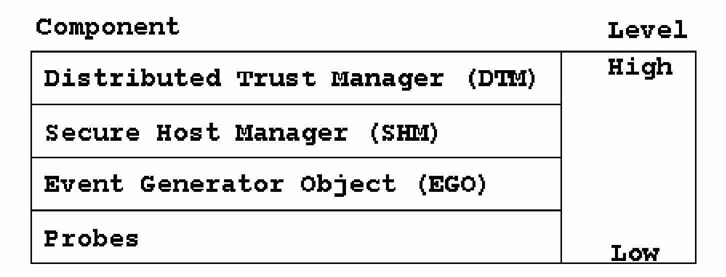


Figure 1. ***S.A.F.E.* Architecture**

S.A.F.E. is composed a set of hierarchical components, as shown in Figure 2 and described in [CO 02].



Component	Level
Distributed Trust Manager (DTM)	High
Secure Host Manager (SHM)	
Event Generator Object (EGO)	
Probes	Low

Figure 2. ***S.A.F.E.* Component Hierarchy**

Each component provides services to the components above them in the hierarchy. The lowest level component is the probes. Above the probes is the Event Generator Object (EGO). Above the EGO is the Secure Host Manager (SHM), and above the SHM is the highest-level component in the hierarchy, the Distributed Trust Manager (DTM). The probes gather information and events from the system by monitoring the output of a variety of intrusion detection systems and log files, such as *Snort*, *Tripwire*, and the *syslog* file on Linux. The probes then provide this data to the EGOs. The EGOs function as a data abstraction layer. They collect the data from the probes, filter it, convert it into a platform independent format, and provide the aggregated, filtered data to the SHM, by putting the data into the Event Queue. In addition to this, the EGOs start and stop the probes.

The SHM provides intrusion detection for the host it monitors. Part of the SHM, the Analysis Engine pulls the data from the Event Queue, processes it, and determines the probability that an intrusion has occurred based on the data. The probability falls into one of three ranges. In the lowest range, the SHM is sure that the data does not represent an attempt to compromise the host. In the middle range, the SHM is unsure whether the data represents an attempt to compromise the host. In the highest range, the SHM is sure the host has been compromised. The probability generated by data is sent to the DTM for further evaluation. If the probability is in the third range, the SHM also sends a "last gasp" message to the DTM. Once it sends the "last gasp" message, the SHM stops the EGOs, stops processing data from the EGOs, and launches countermeasures to restore the node to a healthy state. The SHM also provides the ability to learn from previous attacks and non-attacks to improve its intrusion detection capabilities. The SHM uses information it gets from new data sets that it retrieves from the EGOs to modify the rules it uses to generate the corresponding probabilities to make them more accurate.

The DTM's purpose is to create and maintain a network of trusted computers. Because computers can be compromised and their status can change over time, a computer can be removed from a Trust Domain and new computers are allowed to join a Trust Domain. If any DTM receives a probability from an SHM that is in the second or third ranges, it shares this information with the other DTMs. If the value was in the third range, the computers in the Trust Domain remove the computer that has been compromised from the Trust Domain. If the value was in the second range, all the DTMs test the host that sent the value. If the host passes the tests, it is deemed uncompromised and nothing happens. If the host fails the tests, it is deemed compromised and removed from the Trust Domain.

S.A.F.E. is a real time, distributed system based on the concept of autonomous agents, similar to the architecture proposed in the AAFID system [BA 98]. Prior research on *S.A.F.E.* addressed the first two problems. The research presented in this thesis demonstrates that it is possible to design and implement a distributed IDCS that imposes minimal overhead on the computers in the system and tolerates hardware and software failures. In addition to this, this research demonstrates that a distributed IDCS can be built such that compromising fewer than half the nodes participating in the system does not prevent the IDCS from functioning correctly. *S.A.F.E.* conceptually addresses all the problems that currently plague available IDCSs, and it is the first IDCS we know of that does so.

1.5 Our Work

The focus of this research was to design a DTM that addressed the third and fourth criteria for robust IDCSs and could be seamlessly integrated with the existing architecture of *S.A.F.E.* This would complete the framework for *S.A.F.E.*, making the *S.A.F.E.* framework the first we know that would address all four criteria required to build a robust IDCS. In order to do this, we felt it was critical that the DTM must be a distributed system, so there would be no single point of failure that could bring down the entire system. In order to make the DTM fault tolerant and able to withstand attacks, we examined how other fault tolerant systems had been built in the past, to learn what techniques could be used.

1.6 Fault Tolerant Systems

Because of the frequency of hardware and software failures and the consequences resulting from those failures, systems that are able to tolerate such failures, called fault tolerant systems, became important. The key to building fault tolerant systems is to design them in such a way that some small number of failures does not cause the entire system to fail in such a way that the system cannot recover from the failures. Redundant hardware can be used to provide fault tolerance for a system, where individual devices fail. For example, while a computer with only one hard disk will lose all its data and be unable to function if the hard disk dies, a computer with RAID can reconstruct all the data and continue to function even if one of the hard disks in the array dies. An important concept is that if there are n of one type of component, a properly designed system will be able to tolerate n-1 of those components failing and still operate correctly. Many different fault tolerant systems have been built. For example, VAXclusters have addressed the problem of fault tolerance using redundant hardware [SN 91]. Lampson discusses how to create fault tolerant systems using deterministic finite automata [LA 96]. However, sometimes just having redundant hardware is not enough to make a system fault tolerant. In some cases, a part of the system will fail in such a way that the system cannot detect that it has failed, and thus two redundant parts of the system will have a different state. When the system then needs to determine what the state is, it will be unable to do so. This problem requires a mechanism to arrive at a consensus of what the correct state is. Once consensus has been achieved, the redundant parts can be synchronized and faulty ones can be removed and replaced, if necessary, allowing the system to continue functioning. In their paper, "The Byzantine Generals Problem", Lamport et al cast the problem of achieving consensus as an easily understandable military problem [LL 82]. In the context of this problem, Lamport et al explain how the uncompromised participants can achieve consensus using two different algorithms [LL 82]. These consensus algorithms can be used to build fault tolerant hardware and software systems [LL 82]. We concluded that one of the solutions to the well-known Byzantine Generals Problem was the most appropriate one to use, and we designed and implemented the DTM using this algorithm.

1.7 The Byzantine Generals Problem

The Byzantine Generals Problem is a well-known problem where generals commanding groups of one army have surrounded the enemy army, and the generals must decide what to do [LL 82]. However, the only means they have of communicating with each other is by using messengers [LL 82]. Complicating the situation is the possibility that some of the generals may actually be traitors [LL 82]. According to Lamport et al, the generals need "an algorithm to guarantee that

A. All loyal generals decide upon the same plan of action.

The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition A regardless of what the traitors do.

The loyal generals should not only reach agreement, but should agree upon a reasonable plan. We therefore also want to ensure that

B. A small number of traitors cannot cause the loyal generals to adopt a bad plan" [LL 82].

Condition B is not addressed, since a small number of traitors can only cause an action to be chosen if the other generals are almost evenly split, in which case, it is not fair to conclude either action is bad [LL 82].

To solve the problem and achieve consensus, there needs to be an algorithm that always works, and in order to satisfy the first condition, each of the loyal generals must have the same information [LL 82]. Lamport et al further define the problem as

"Byzantine Generals Problem. A commanding general must send an order to his n-1 lieutenant generals such that

IC1. All loyal lieutenants obey the same order.
IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends" [LL 82].

Lamport et al refer to IC1 and IC2 as "interactive consistency conditions", and observe that when the commanding general is loyal, "then IC1 follows from IC2" [LL 82].

Lamport et al devised two solutions to this problem – the Oral Message algorithm and the Signed Message Algorithm [LL 82]. Each of these algorithms has advantages and disadvantages. The Oral Message algorithm requires that more than 2^n messages be sent to achieve consensus, if there are n generals, and it only works if the number of loyal generals is more than twice the number of traitorous generals [LL 82]. The Signed Message Algorithm, in contrast, only requires sending $O(n^2)$ messages to achieve consensus, and works regardless of the number of traitorous generals [LL 82]. The Signed Message Algorithm requires a greater number of necessary conditions to be met in order to operate correctly than the Oral Message Algorithm does. However, it is significantly faster than the Oral Message Algorithm, and the extra requirements are not problematic in our system, so we elected to use the Signed Message Algorithm. We explain the Signed Message Algorithm fully in section 2.

1.8 Terminology

Throughout this paper, we will refer to some terms, which we define here. We will also use the words **computer**, **host**, **node**, and **system** interchangeably. We say that a computer is **compromised** if the *S.A.F.E.* software (described later) installed on it is not functioning correctly. A computer that is not compromised will be referred to as **uncompromised**. A **Trust Domain** is a set of computers that can communicate with each other and believe that each other are not compromised. A **signed message** is a piece of data that is created and then signed by the creator in such a way that the contents of the message cannot be changed without the recipient of the message being able to tell that the contents were changed. The signature of the message creator, $host_i$, is a set of text that can only be created by host$_i$. However, every other host$_z$ can verify whether a message was signed by host$_i$. A message that is **forged** is a message that is created by a host$_i$ and then signed so the message appears to have been created by host$_z$. A message that has been **tampered with** is a message whose contents were created by host$_i$ and then changed by some other host, host$_z$. A **signature** is a sequence of some number of events, e_1, e_2, ..., e_n, that represent a possible attack on a computer system. Each **event** is a single action, such as an attempt to login to a computer or an attempt to gain administrative privileges on a system. The events in a signature need only to occur in the correct order, but not immediately following each other, to represent a potential attack. This is because some attacks happen over time, and because other users may make legitimate requests of the system in between the events in an attack.

2 The DTM

The DTM is responsible for forming and maintaining Trust Domains. A Trust Domain is a set of nodes that share a charter and a security policy, and which behave consistently and according to the security policy. A node can be any device with a sufficient amount of computing capability to carry out the necessary tasks of the DTM. Thus, although our work has been with Trust Domains composed of Intel®-based personal computers, we envision Trust Domains composed of many different devices, including workstations, routers, and even members of a distributed database. The nodes in the Trust Domains work together to keep compromised nodes from joining trust domains, and to detect and remove any nodes that become compromised after joining the Trust Domain.

The objectives of the DTM are analogous to those of the generals in the Byzantine Generals Problem. The nodes in the Trust Domain are analogous to the generals. The nodes must work together to decide whether to allow another node to join the Trust Domain, while the generals must collaborate to decide whether to attack or retreat. There may be nodes in the Trust Domain that are compromised, just as there may be some generals that are traitors. We also assume that a node that does not participate in the process of achieving consensus is compromised. Thus, a node in the Trust Domain that experiences a hardware or software failure and is unable to participate in the process is analogous to a general who does not participate in the process of achieving consensus either by not sending messages or because his messenger was prevented from reaching the other generals. In addition to just concluding whether to allow a node to join a Trust Domain, the nodes in the Trust Domain must also discover and remove any compromised nodes, or traitors. This is an extension to the Byzantine Generals Problem, but due to the similarities between how the DTM and the generals must achieve consensus, we believed that we could adapt a solution to the Byzantine Generals Problem for use by the DTM. However, in addition to these objectives, there are some other requirements that the DTM must satisfy to be usable.

2.1 Requirements

There are many requirements that the DTM must satisfy in order to be effective. The DTM must protect itself from attacks, or else an attacker could disable *S.A.F.E.* like many other IDCSs and render it useless. The DTM protects itself by trying to maintain at least a simple majority of uncompromised hosts in a Trust Domain. The hosts in a Trust Domain decide whether to admit new hosts or remove current hosts from the Trust Domain. In order to do this, the DTM needs all the uncompromised hosts to agree on the actions to take. Thus, the DTM needs to use a consensus algorithm.

The DTM must be fault tolerant in order to work in a realistic environment. We define a fault as a host in a Trust Domain going offline because of a hardware, software, or power failure, a host becoming unreachable because of a network failure or congestion, or a host being compromised by an attacker.

It is important that the DTM not have a central server or component because a central server would be a weak link in the armor of *S.A.F.E.* A central server could be attacked and either compromised or subjected to a denial of service attack, rendering *S.A.F.E.* inoperable and useless.

It is crucial that an implementation of the DTM be neither CPU intensive nor network intensive. If the DTM consumes too much CPU time or network bandwidth, then it will not be useful because it will prevent the system from performing its normal tasks or will take too long to allow new systems to join a Trust Domain.

S.A.F.E. must be scalable to allow enough computers to join Trust Domains and communicate with each other so that a large number of computers can talk to each other and determine whether another computer is trustworthy.

When a computer, c_i, tries to join a Trust Domain or a computer already in a Trust Domain is suspect (ways of determining a suspect node will be described later in section 2.3.3), the DTMs running on the other computers in the Trust Domain send signatures to c_i. Each signature represents a false positive, a false negative, or an actual attack and has a probability associated with it. The probability is the likelihood that the signature (regardless of whether it is a false positive, false negative, or an attack) is an attack. Each computer that is not compromised knows the probability that corresponds to the signatures it sends. If the *S.A.F.E.* software on c_i is not compromised, then for each signature it receives, c_i will be able to determine and return the correct probability that the signature was an attack to the sender of the signature. If the *S.A.F.E.* software on c_i is compromised, then c_i should be unable to determine the correct probability except with rare exceptions where it makes a lucky guess. Therefore, the DTMs must prevent compromised computers from being able to collect the data the DTMs use to test if a host has been compromised. This is discussed in more detail in the future work section.

2.2 The Signed Message Algorithm

2.2.1 Assumptions

We claimed that a variant of a solution to the Byzantine Generals Problem, specifically Lamport et al's Signed Message Algorithm, would allow the DTM to achieve consensus. There are several assumptions that are required for the Signed Message Algorithm to work [LL 82]. The assumptions are:

- All messages are delivered to the intended recipient.
- The recipient of a message can tell who sent it.
- The absence of a message can be detected.
- Any tampering with a message can be detected and no loyal general's signature can be forged.
- Anyone can verify the authenticity of a general's signature. [LL 82].

2.2.2 Formal Definition Of The Signed Message Algorithm

Before defining the Signed Message Algorithm formally, Lamport et al define a function *choice*(V), where V is a set [LL 82]. "If the set V contains a single element *v*, then *choice*(V) = *v*" [LL 82]. If V = Ø, then *choice*(V) returns a default value, and *choice*(v_1, v_2, ..., v_i) for $i \geq 2$, can be defined to deterministically return any of the v_i or the default value [LL 82]. Lamport et al formally define the Signed Message Algorithm as

"Algorithm SM(m).

```
Initially, V_i = Ø.
  (1) The commander signs and sends his value to every lieutenant.
  (2) For each i:
        (A) If Lieutenant i receives a message of the form v:0 from the commander
        and he has not yet received any order, then
            (i) he lets V_i equal {v};
            (ii) he sends the message v:0:i to every other lieutenant.
        (B) If Lieutenant i receives a message of the form v:0:j_1:...:j_k and v is not in
        the setV_i, then
            (i) he adds v to V_i;
            (ii) if k < m, then he sends the message v:0:j_1:...:j_k:i to every lieutenant
                 other then j_1,...,j_k.
  (3) For each i: When Lieutenant i will receive no more messages, he obeys the order
choice(V_i)" [LL 82].
```

2.2.3 Explanation Of How The Signed Message Algorithm Works

Perhaps one of the best features of the Signed Message Algorithm is that it functions even if some of the hosts in the system are compromised. In fact, for the Byzantine Generals Problem, the algorithm works even if all but one of the hosts are compromised, and it works trivially if all of the hosts are compromised. This is not sufficient for *S.A.F.E.*, however, and we give more restrictive assumptions about the number of compromised hosts in sections 2.3.2 and 2.3.3. In the Signed Message Algorithm, one of the hosts acts as a leader and sends an order to the other hosts. Whenever a host receives a message, it checks the message to determine if the message has been forged or tampered with. The DTM is able to determine whether a message has been forged, tampered with, both forged and tampered with, or neither. However, the DTM is unable to gain any additional information beyond this, so it treats any message that has been forged, tempered with, or both, in the same manner, and discards it without doing any further processing. If the message has not been forged or tampered with, the receiver takes the order and puts it in its list of orders received. Then the receiver signs the message with his own signature and forwards it to all the hosts whose signature is not on the order. If a host receives a message with an order that is already in his list, he ignores the message. When no more messages will be received, the hosts all choose an order from the list of orders they have received using this method. If only one order has been received, that order is chosen. If more than one order has been received, a previously designated default order is chosen.

Because any order that reaches a loyal general will be forwarded to all other generals who have not seen the order, all the loyal generals will have the same set of orders to choose from, and thus choose the same order to obey. While the Signed Message Algorithm satisfies the requirements to achieve consensus, if the "commander" from Lamport et al's Signed Message Algorithm is compromised, it may cause the wrong answer to be the one reached by consensus because the uncompromised hosts will all agree on whatever order the leader sends. This clearly does not satisfy our requirement that uncompromised hosts must be admitted into the Trust Domain and compromised hosts must not be admitted into the Trust Domain. If the Signed Message Algorithm can be used to accomplish our goals, it must be modified.

2.2.4 Theorem: The Signed Message Algorithm Solves The Byzantine Generals Problem

Lamport et al present the following theorem:

"Theorem 2. For any m, Algorithm SM(m) solves the Byzantine Generals Problem if there are at most m traitors" [LL 82].

In order to prove this, one just needs to prove that both IC1 and IC2 hold [LL 82]. Lamport et al begin by proving IC2 holds [LL 82]. There are two possible cases. The commander can either be loyal or a traitor. We note that IC2 from section 1.7 can be rewritten as $p \rightarrow q$. Thus, if the commanding general is not loyal, then $p \rightarrow q$ is satisfied trivially. Therefore, we must just show that if the commanding general is loyal, then all the loyal lieutenants obey the commanding general's order [LL 82]. We assume the commanding general is loyal [LL 82]. Then in step 1 of SM(m), the commanding general sends the signed order v:0 to each of the other generals [LL 82]. In step 2A, every general will receive the order v. Since no general can forge messages, no general will be able to receive any other order v':0 in step 2B [LL 82]. Thus, in step 3, each general will only have one order to choose from, so *choice*(V_i) will return the same order, *v*, for each general *i* [LL 82]. Thus, all the loyal generals will follow order *v*, which is the order the commanding general sent, and thus, IC2 is satisfied [LL 82].

Now, the only thing remaining is to prove that IC1 always holds [LL 82]. As stated in section 1.7, if IC2 holds and the commanding general is loyal, then IC1 follows from it, so it is sufficient to show that IC1 holds when the commanding general is not loyal [LL 82]. We assume the commanding general is not loyal. Since the choice function always returns the same value if $V_x = V_y$, then we must just show that for any two loyal generals x and y, that $V_x = V_y$ [LL 82]. This amounts to showing that if general x receives a correctly signed order o in step 2, then general y receives the correctly signed order o in step 2 [LL 82]. Assume that loyal general x receives the correctly signed order o in step 2A [LL 82]. Then general x will sign the message and forward it on to all other generals, so general y will receive the order [LL 82]. Assume that general x received the correctly signed order o in step 2B [LL 82]. Then, since forged or tampered messages can be detected, and are thus discarded, either general y has already received the message and has signed the message that was sent to general x, or else general x sends the message to general y because general x can tell that general y has not signed the message [LL 82].

2.3 How The DTM Works

2.3.1 General Concept

We have established a basic algorithm for the DTM to use in order for the DTM to satisfy the given requirements. When a new host tries to join a Trust Domain, it sends a message to the members of the Trust Domain. The members of the Trust Domain all verify that the new host is running an operating system that is updated with any required patches and has the latest version of the *S.A.F.E.* software installed. If the new host meets these minimum requirements, the members of the Trust Domain interrogate the new host to ensure that the *S.A.F.E.* software has not been compromised. The results of these interrogations are then shared with the other members of the Trust Domain and the members of the Trust Domain reach a consensus on whether to admit the new host into the Trust Domain. Finally, as described in section 2.3.3, the DTM tries to detect and remove any hosts that have become compromised since they joined the Trust Domain. The DTM needs a method of achieving consensus, and we believe that this may be done using a variant of a solution to the Byzantine Generals Problem. We also must make some assumptions, to get the initial version of the DTM implemented.

2.3.2 Assumptions For The DTM

In order for the concept of a Trust Domain to work, we must assume that it can be started with an uncompromised host. We assume that a system administrator starts the *S.A.F.E.* software on a host that is known to be uncompromised. We also assume that the system administrator starts the *S.A.F.E.* software on two additional uncompromised hosts and has them join the Trust Domain. These first 3 computers form the foundation of the Trust Domain. We also assume that compromised hosts cannot return the correct probability corresponding to signatures sent to them by uncompromised DTMs, except in extremely rare instances. The existence of a scheme to keep the data used to test if computers are uncompromised is assumed. Such schemes are discussed in the future work section. We also assume that a majority of hosts in a Trust Domain are not compromised. We define a simple majority of hosts in a Trust Domain to be 1 + Floor(n/2), if there are n hosts in a Trust Domain. This is a critical assumption. *If the number of compromised hosts in a Trust Domain exceeds the number of uncompromised hosts, then the compromised hosts have taken over the Trust Domain. In this case, the compromised hosts can prevent the uncompromised computers from being able to restore the Trust Domain to a state where the majority of the computers are uncompromised.* We provide a mechanism to attempt to prevent the number of compromised systems from growing too large, by trying to detect and remove compromised hosts from the Trust Domain.

2.3.3 Parallel Implementation of the Signed Message Algorithm

Running n instances of the Signed Message algorithm in parallel allows us to detect compromised hosts within the trust domain, as described below, and it achieves consensus about whether to let a host join the trust domain. This method requires $O(n^3)$ messages to be sent, where n is the number of hosts in a Trust Domain. By limiting the number of hosts in a Trust Domain to 16 at any time, $O(n^3)$ is a manageable number of messages, as demonstrated by the resource utilization data in section 4.1. Scalability of *S.A.F.E.* to allow it to function in a network of several thousand computers is discussed in section 6.

In addition to just achieving consensus which one execution of the Signed Message Algorithm can do by sending $O(n^2)$ messages, the parallel version guarantees that compromised nodes are not admitted to a trust domain, which can happen if the decision is based on one execution of the Signed Message Algorithm. **The guarantee is only valid as long as a simple majority of the hosts in the trust domain are not compromised, which is one of our assumptions.**

Compromised hosts in a trust domain are detected by using the parallel version of the Signed Message Algorithm, assuming that a majority of the hosts within the trust domain are not compromised. This result is because there are only a limited number of possible events that can occur when the Signed Message Algorithm is run.

If the "commander" from Lamport et al's Signed Message Algorithm is not compromised, then after running the Signed Message Algorithm in parallel, all hosts that are not compromised know that the leader is not compromised, because the leader sent the correct conclusion that all hosts that are not compromised determined.

If the leader of an execution of the Signed Message Algorithm is compromised, then it can send:

- 0 messages to all uncompromised nodes - All uncompromised nodes assume node compromised or dead (same treatment as compromised).

- 1 message to only some uncompromised nodes – The uncompromised nodes detect there's a compromised node in the system and send a message to other nodes accusing the suspected compromised nodes. The accused nodes are then tested and are determined to be compromised or uncompromised.

- 1 message to all uncompromised nodes - All uncompromised detect message is wrong and host is compromised if it contradicts the majority. If message does not contradict the majority, this cannot be detected yet unless the node sends a different message to at least 1 compromised node that forwards the message to at least 1 uncompromised node. Although this node should be removed, it is not currently causing damage, so it is not a critical problem.

- 2 (or more) different messages to some uncompromised nodes - All uncompromised nodes see contradictory instruction and know node is compromised.

2.4 Denial Of Service Attack Prevention

As stated earlier, a requirement of the DTM is that it consumes few enough system resources, such that it does not prevent the system it's protecting from operating normally. If DTM is constantly checking hosts that are trying to join a Trust Domain, then it will be using a lot of CPU time and network bandwidth. Therefore, a compromised host that continuously tries to join a Trust Domain will cause all the systems in the Trust Domain to spend significant amounts of CPU time and network bandwidth needlessly. In order to prevent this, DTM has a set of configurable parameters. These parameters allow the system administrator to specify how many attempts a host can make to join a Trust Domain in an amount of time, such as three attempts in 5 minutes, before additional requests from that host are considered a denial of service attack. Once a request to join a Trust Domain is deemed a denial of service attack, all subsequent requests to join made by that host are ignored for a specified amount of time, which is also configurable by the system administrator.

The implementation of the DTM is such that an uncompromised node will not send multiple join requests in rapid succession. When an uncompromised node tries to join a Trust Domain, it sends a single join request using UDP and waits for a response. The only reason that there would be no response is if the network is unreachable due to a severed communication link, the UDP packet was not received, or the network is so congested that a single UDP packet gets stuck in a queue and is not processed for a long time. If there is no response in a configurable amount of time, the *S.A.F.E.* software terminates because it assumes that it will not receive a response. In this case, the administrator will have to start the *S.A.F.E.* software again to have the computer retry to join the Trust Domain. If there was no response because of a severed communication link or because the packet was not received, then restarting the *S.A.F.E.* software and trying to join the Trust Domain again will not cause multiple requests to be sent because the previous ones were never received. If the request was not received in time because it sat in a queue too long, then the network is likely horribly congested and the administrator will be likely busy trying to relieve the congestion and not restarting the *S.A.F.E.* software, as *S.A.F.E.* will not be a priority in that case.

3 Experiments

In our experiments, it was unnecessary to actually **attack** and **compromise** computers or the software running on them. We could test both the DTM and *S.A.F.E.* by emulating attacks on computers and having them emulate being compromised. Because of this, we did not actually attack or compromise nodes. Therefore, it should be understood that our results do not conclusively **prove** that our implementation works. However, our results should provide a good indication of whether the implementation worked. We still use the words *attack* and *compromise* in our descriptions of the experiments, results, and analysis, but it should be understood that we mean an emulated attack and an emulated compromise.

3.1 Rationale For Design Of Experiments

3.1.1 Trust Domain Model

We have constructed a model, $M = \{C, C_{new}, P, X, \alpha\}$, for a Trust Domain. This model allows us to reason about the correctness of the Distributed Trust Manager (DTM). One potential side benefit of the model is that it will allow us to identify holes in our design validation and testing of the DTM. The basic components of our model, C, P, X, and α are defined below.

$C = \{c_1, c_2, c_3, ..., c_n\}$, for some finite integer n, is the set of nodes in the Trust Domain. While n could be very large, for practical purposes associated with the implementation of DTM, we will limit n to 16.

$C_{new} = c_j \mid c_j \not\subset C$, which is a node that is trying to join a Trust Domain. The state, w, of C_{new} is unknown at the time that C_{new} declares its intention to join a Trust Domain. That is, C_{new} may be a node whose behavior is in accordance with the Trust Domain admission and security policies ("uncompromised node") or it may be compromised.

$\alpha = \{\alpha_1, \alpha_2, \alpha_3, ..., \alpha_m\}$ is the set of actions that a node may perform. A node is essentially a deterministic finite automaton. Thus, it has a finite set of states, a finite set of input symbols, and a transition function that causes the node to move from some state s_i to some state s_j (s_i may be s_j) based on the input it receives [HM 00]. The transition function is analogous to the set of actions a node may take based on the input the node receives. Since the input alphabet is finite and the number of states is finite, the set of actions, α, which a node may take, is finite.

$P = \{p_1, p_2, p_3, ..., p_k\}$ is a security policy, which describes the acceptable behaviors of a computer in the Trust Domain. Each p_i is a rule composed of a subject, an object, and an action that the subject may perform on the object. We note that security policies are rarely defined completely, as it is difficult to list every possible rule that is acceptable. Thus, we accept that we will have to add new rules to the policy as new acceptable actions that a computer may perform on an object are discovered. It should be clear that $P \subseteq \alpha$. We also note that $\neg P \cup P = \alpha$.

X is a capability matrix, which indicates what actions the subjects can perform on the objects and/or on each other. The subjects here are the nodes. The objects are the data and the services running on the nodes. $X[i][j] = \{(n, \{\alpha_i \in \alpha \mid c_i$ can perform actions α_i to object n on computer $c_j\})\}$. In other words, for each object n on computer c_j, X[i][j] gives the set of actions that computer c_i can do to object n on computer c_j.

We added code to the DTM to allow us to emulate nodes being compromised during the experiments. During the trials, each compromised nodes could assume any of the compromised behavior patterns, which are discussed in the following paragraphs and shown in table 5. There are two characteristics of compromised nodes. The first characteristic is how they behave when they are the "commander" of the Signed Message Algorithm. The second characteristic is how they behave when they are a "lieutenant" in the Signed Message Algorithm.

We define the set of leader behavior patterns to be the set L = {Leader Behaves Normally, Leader Sends No Messages, Leader Sends All Hosts Wrong Messages, Leader Sends Some Hosts Wrong Messages, Leader Sends Some Hosts Right Messages, Leader Sends Some Hosts Wrong & Right Messages, Leader Sends All Hosts Wrong & Right Messages}.

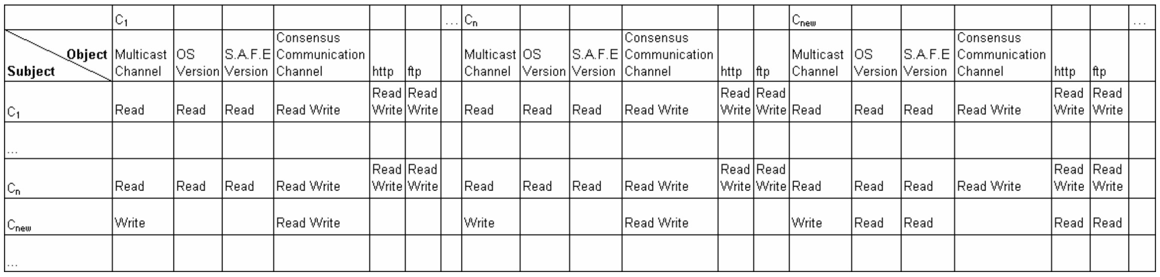
We define the set of follower behavior patterns to be the set F = {Follower Behaves Normally, Forwards No Messages, Forwards Some Messages, Forwards Messages to Some Hosts}.

Since each node will act as a leader in one instance of the Signed Message Algorithm and as a follower in the other instances, the set of possible behavior combinations is the Cartesian product of the sets L and F. The Cartesian product L x F consists of 28 behavior combinations, one of which is the node behaves correctly as a both a leader and follower. Hence, the number of compromised behavior patterns that a single node can exhibit is $(7 \times 4) - 1 = 27$.

3.1.2 Capability Matrix

We constructed a capability matrix, using the subjects, objects, and behaviors in the Trust Domain. The capability matrix was then used to define the security policy, P.

This, in turn, allowed us to construct a test case matrix, which enumerates the test cases we needed to perform.



	C_1						…	C_n						C_{new}						…
Object / Subject	Multicast Channel	OS Version	S.A.F.E Version	Consensus Communication Channel	http	ftp		Multicast Channel	OS Version	S.A.F.E Version	Consensus Communication Channel	http	ftp	Multicast Channel	OS Version	S.A.F.E Version	Consensus Communication Channel	http	ftp	
C_1	Read	Read	Read	Read Write	Read Write	Read Write		Read	Read	Read	Read Write	Read Write	Read Write	Read	Read	Read	Read Write	Read Write	Read Write	
…																				
C_n	Read	Read	Read	Read Write	Read Write	Read Write		Read	Read	Read	Read Write	Read Write	Read Write	Read	Read	Read	Read Write	Read Write	Read Write	
C_{new}	Write			Read Write				Write			Read Write			Write	Read	Read		Read	Read	
…																				

Figure 3 Capability Matrix for Computers in a Trust Domain

3.1.3 Security Policy

We define several variables and constants that are necessary to define the security policy, P. The variables are:

os_i = the operating system version on computer i
$safe_i$ = the *S.A.F.E.* version on computer i
$response_{i,x}$ = the response computer i gives to computer x in the Trust Domain when it is interrogated
$opinion_{i,x}$ = the belief of computer x in the Trust Domain about whether computer i is compromised
$majority_{i,x}$ = the majority belief of the computers in the Trust Domain about whether computer i is compromised after the parallel Signed Message Algorithm has been completed
$messageTampered_i$ = whether message i has been tampered with

The constants are:

os_T = the minimum acceptable version of the operating system for a node to join Trust Domain T.
$safe_T$ = the minimum acceptable version of *S.A.F.E.* for a node to join Trust Domain T.
$response_c$ = the correct response computer to an interrogation
compromised = the majority belief that the computer trying to join the Trust Domain is compromised
uncompromised = the majority belief that the computer trying to join the Trust Domain is not compromised

p_1: Given a node, $c_i \in C_{new}$, that is attempting to join a Trust Domain, T, then, if the value of $os_i < os_T$, c_i is not allowed to join Trust Domain T.

p_2: Given a node, $c_i \in C_{new}$, that is attempting to join a Trust Domain, T, then, if the value of $safe_i < safe_T$, c_i is not allowed to join Trust Domain T.

p_3: Given a node, $c_i \in C_{new}$, that is attempting to join a Trust Domain, T, and a node c_x in Trust Domain T, then, if the value of $response_{i,x}$!= $response_c$, then $opinion_{i,x}$ = compromised (c_i is deemed compromised by c_x).

p_4: Given a node, $c_i \in C_{new}$, that is attempting to join a Trust Domain, T, and a node c_x in Trust Domain T, then, if the value of $response_{i,x}$ == $response_c$, then $opinion_{i,x}$ = uncompromised (c_i is deemed uncompromised by c_x).

p_5: Given a set of nodes, c_1, ..., c_x, in a Trust Domain T, a node c_i not in Trust Domain T, and $majority_{i,x}$ == compromised, c_x does not open a socket connection to c_i (thus not allowing c_i is to join Trust Domain T).

p_5: Given a set of nodes, c_1, ..., c_x, in a Trust Domain T, a node c_i not in Trust Domain T, and $majority_{i,x}$ == uncompromised, c_x opens a socket connection to c_i (thus allowing c_i is to join Trust Domain T).

p_6: Given nodes c_x and c_y in Trust Domain T, and c_x receives message i from c_y, such that $messageTampered_i$ == true, c_x discards message i as if it was never received.

3.1.4 Test Case Matrix

We have several test objectives. Our first objective is to ensure that computers in a Trust Domain behave in accordance to the security policy, P, and the capability matrix, X. We are trying to ensure that all the behaviors in $\neg$P are not allowed. Finally, we seek to update P and $\neg$P continually, moving all actions from $\neg$P to P, if the actions are not destructive to the functioning of the Trust Domain. In figure 4, we give a test case matrix designed to demonstrate a set of test cases that must be run to verify that the DTM functions correctly. However, the matrix is intended to serve as an example only. It is not complete and thus not sufficient.

	No Hosts in Trust Domain are Compromised	Some Hosts in Trust Domain are Compromised
Keeps Compromised Hosts From Joining		
Detect and Remove Nodes that Become Compromised After Joining		
Stop Repeated Attempt to Join DOS Attack		
Allow Uncompromised Nodes to Join		
Measure Resource Utilization		

Figure 4 Test Case Matrix

3.2 Experimental Setup

The hardware used to evaluate both the DTM and *S.A.F.E.* was a set of 10 computers with the following CPUs:

- 5 Intel® Pentium® 3 450 MHz processors
- 3 Intel® Pentium® 2 400 MHz processors
- 1 Intel® Celeron® 466 MHz processor
- 1 Intel® Celeron® 400 MHz processor

All of the computers had 128 MB of RAM, except one of the computers with an Intel® Pentium® 3 450 MHz processor, which had 320 MB of memory. Each computer was directly connected to a 16-port NETGEAR EN116 10BASE-T Ethernet hub. Neither the hub nor any computer was connected to any other network. All but one computer ran the Debian Linux version 3.0 operating system without a window manager, the J2SE 1.4.2_04 from Sun Microsystems, Snort 1.8.4-beta1, and *S.A.F.E.* 1.0. The other computer, which was the one with 320 MB of RAM, ran Microsoft Windows 2000, the J2SE 1.4.1_02 from Sun Microsystems, and *S.A.F.E.* 1.0. This computer ran the GUI that was used to test the DTM. In a production environment, no GUI would be necessary.

3.3 Experiments to Evaluate the DTM

Due to time constraints, it was clear that only a subset of the test cases could be run, and they could only be run a limited number of times. Because of this, we acknowledge that the testing was not as thorough as it should be and the results are not statistically significant. Therefore, the results will only give a rough indication as to the feasibility of the DTM. We chose to test these features of the DTM, as they were the ones the model and requirements indicated were crucial.

1. Whether uncompromised hosts were allowed to join the Trust Domain.
2. Whether compromised hosts were not allowed to join the Trust Domain.
3. Whether hosts that became compromised after joining the Trust Domain were detected and removed from the Trust Domain based on information gained when new hosts tried to join the Trust Domain.
4. Whether a compromised host was prevented from being able to cause a Denial of Service attack by repeatedly trying to join the Trust Domain after being rejected.
5. What the CPU utilization of the systems in the Trust Domain was just to run the DTM.

To test the effectiveness of the DTM, a test harness was used. This allowed us to emulate computers becoming compromised using a GUI. The "compromised" computers behaved in one of the 27 compromised computer behavior patterns given by the model. The behavior of each compromised computer was chosen randomly from the set of possible behaviors, using the GUI.

The resource utilization of the DTM was measured on an unloaded system that was only running the *S.A.F.E.* software, *Snort*, and *top*, a performance-monitoring tool. First, *top* was started and began to produce output. Then *S.A.F.E.* was started. *top* measured the resource utilization of the computer every second and printed the results to a log file. The log file was analyzed later, and the maximum amount of memory that the DTM used was calculated by subtracting the amount of memory used when *S.A.F.E.* was not running from the maximum amount of memory used when *S.A.F.E.* was running. The percent of CPU utilization was calculated by averaging the amount of CPU utilization while *S.A.F.E.* was running. The measurements were recorded for a computer with an Intel® Pentium® 3 450 MHz processor, which was not running the GUI.

3.4 Experiments to Evaluate S.A.F.E.

In order to evaluate *S.A.F.E.*, its performance was compared to that of *Snort* 1.8.4-beta1. Using *Snot 0.92a*, 10 signatures were sent to a computer running *Snort*, to see how *Snort* would respond to the signatures. Next, a Trust Domain of nine computers running *S.A.F.E.* was formed, and three nodes were selected at random to emulate the behavior of a compromised node. Then a signature was sent to one of the uncompromised nodes and the results were observed and recorded. This procedure was repeated 3 times for each signature. In each trial, the behavior of the compromised nodes was selected by having a random number generator pick the behavior from the set of possible behaviors. In these trials, when the SHM received the signatures, if they represented a likely attack, the SHM sent a message to the DTM. After that, if the signature was an attack, the SHM was set to behave as that particular node was compromised instantaneously.

The 10 signatures that were used were determined by members of the WPI System Security Research Lab Research Group, in an attempt to create realistic attacks. The signatures were designed to emulate attacks on a computer. Each signature falls into one of three categories. Signatures are either clearly not an attack, clearly an attack, or it is unclear whether they are an attack. A low threshold and a high threshold separate the different types of signatures. Three of the signatures emulated an unsuccessful attack, and thus, they were assigned a probability less than the low threshold. Three of the signatures emulated a successful attack, and were assigned a probability greater than the high threshold. Four of the signatures emulated an attack that might be a successful attack or a failed attack. These were assigned a probability between the low and high thresholds. The assigned probabilities were .1, .9, and .5 respectively, which are arbitrarily chosen values that fall within the appropriate ranges When the signatures that were not definitely failed or successful attacks were sent, the tests were done twice. The first time they were done as if the attack had not been successful. The second time they were done, the SHM on the computer that had been "attacked" emulated being compromised.

4 Experimental Results and Analysis

4.1 Results

The results of the experiments are shown in Tables 1, 2, 3, and 4 and Chart 1. Table 1 shows how successful the DTM was for each test.

	Hosts in Trust Domain			
Task	Compromised	Uncompromised	Successes	Failures
1. Keeps Compromised Hosts	4	5	10	0
From Joining	5	4	8	2
2. Detect and Remove Nodes That Become	4	5	10	0
Become Compromised After Joining	5	4	7	3
3. Stop Repeated Attempt to	4	5	5	0
Join DOS Attack	5	4	5	0
4. Allow Uncompromised	4	5	5	0
Nodes To Join	5	4	4	1

Table 1 Results of Testing the DTM

Table 2 gives the approximate resource utilization of the DTM during 5 trials.

Trial	RAM consumed by *S.A.F.E.* (MB)	Average CPU Utilization (%)
1	21.527	6.9
2	21.441	6.9
3	20.949	6.9
4	21.672	7
5	20.91	6.9

Table 2 - Resource Utilization of the DTM

Table 3 shows the approximate resource utilization of *S.A.F.E.* when it was run for a Trust Domain of 9 nodes and was attacked with each of the signatures.

Signature	Probability of Compromise	Result of Signature	trial 1		trial 2		trial 3	
			CPU (% idle)	RAM Used (K)	CPU (% idle)	RAM Used (K)	CPU (% idle)	RAM Used (K)
1	0.9	Compromised	92.1	21416	92.2	21408	92.1	21384
2	0.9	Compromised	92.1	21256	92.1	21136	92.1	21348
3	0.9	Compromised	92	21224	92.2	21668	92.1	21508
4	0.5	Not Compromised	91.9	21656	91.9	21716	92	21696
5	0.5	Not Compromised	91.9	21588	91.9	21724	91.9	21684
6	0.5	Not Compromised	91.9	21720	91.9	21536	91.9	21552
7	0.5	Not Compromised	92	21816	91.9	21902	91.9	21932
4	0.5	Compromised	92.2	21868	92.2	21700	92	21844
5	0.5	Compromised	91.9	21804	91.9	21768	91.9	21728
6	0.5	Compromised	91.8	21700	91.8	21724	91.9	21724
7	0.5	Compromised	92	21508	91.9	21768	91.9	21736
8	0.1	Not Compromised	92.279	21328	92.2	21324	92.3	21528
9	0.1	Not Compromised	92.12	21376	92.258	21512	92.1	21448
10	0.1	Not Compromised	92.384	31428	92.276	21728	92.32	21320

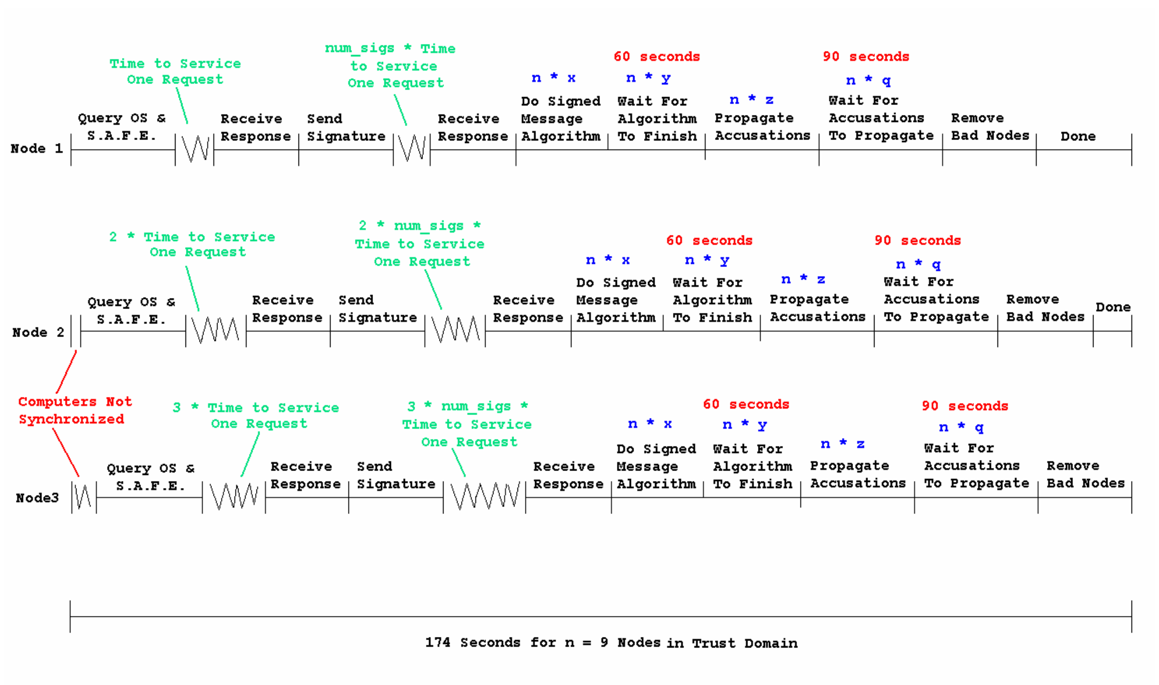
Table 3 - *S.A.F.E.*'s Resource Utilization

Table 4 shows the approximate time it took for the computers in the Trust Domain to determine whether the *S.A.F.E.* on the node that was attacked was compromised.

Signature	Result of Signature	Trial 1 Time (msec)	Trial 2 Time (msec)	Trial 3 Time (msec)
8, 9, 12	Not Compromised	170406	173529	169536
11, 9, 12	Not Compromised	174263	174758	170549
12, 11, 1	Not Compromised	173635	173193	172305
12, 12, 9	Not Compromised	173237	173332	172508
8, 9, 12	Compromised	176552	173503	176373
11, 9, 12	Compromised	175527	175890	176569
12, 11, 1	Compromised	172487	173601	175604
12, 12, 9	Compromised	173452	176827	177449

Table 4 - Time to Determine Whether an Attack Compromised the *S.A.F.E.* Software

Chart 1 is a high-level illustration of how the DTM spends the time it takes to determine if a node is compromised. For clarity, only 3 of the 9 nodes in the system are shown.



Time to Service One Request
num_sigs * Time to Service One Request
60 seconds
90 seconds
n * x
n * y
n * z
n * q
Node 1
Query OS & S.A.F.E.
Receive Response
Send Signature
Receive Response
Do Signed Message Algorithm
Wait For Algorithm To Finish
Propagate Accusations
Wait For Accusations To Propagate
Remove Bad Nodes
Done
2 * Time to Service One Request
2 * num_sigs * Time to Service One Request
Node 2
Computers Not Synchronized
3 * Time to Service One Request
3 * num_sigs * Time to Service One Request
Node3
174 Seconds for n = 9 Nodes in Trust Domain

Chart 1 – How the DTM's Time is Spent

4.2 Analysis

In order to interpret the results of testing the DTM correctly, it is crucial to understand the model of a compromised node that was used in the tests. Two models of compromised nodes were used in the testing. In the unit testing of the DTM, a compromised node consisted of an uncompromised SHM and a compromised DTM. In the testing of *S.A.F.E.*, the model of a compromised node consisted of a compromised SHM and an uncompromised DTM.

The model of a compromised node we used for the testing of the DTM differs from an uncompromised node in one or two ways. Each node acts as a leader and as a follower in the Signed Message Algorithm. A compromised node is one that does not act correctly as a leader, a follower, or both in the Signed Message Algorithm. In section 3.1.1, the ways that a node may behave as a leader and as a follower are specified. These behaviors allow us to construct the matrix in Table 5, where the entry in the ith row and jth column of the matrix indicate one possible way a compromised node behaves. A node is assigned to behave as a compromised node in the Signed Message Algorithm by randomly selecting the compromised behavior. Once a node is assigned to be compromised, it behaves as specified in table 5, and its behavior does not change with time. Therefore, once a node has been assigned to behave as a compromised node with leader behavior i and follow behavior j, it will not adopt any different behavior in the future. Because of this, once a node has been compromised, it will behave as a compromised node with its specified behavior until it is removed from the Trust Domain or the Trust Domain is destroyed. In addition to this, the compromised nodes in the current model follow the same code as the uncompromised nodes in every other way.

This means the compromised nodes will allow or deny a node to join a Trust Domain based solely on the majority opinion. This does not give the compromised nodes the ability to do anything but the same thing as the majority. In fact, the compromised nodes even count their own vote as it should be counted, even if this contradicts the message they send to other nodes. This is a very restrictive model and is not realistic, but it is well beyond the scope of this thesis to develop a better model, and it is left for future students to do. We do, however, provide some suggestions for what a more accurate model should allow.

A more accurate model should allow compromised nodes to behave any possible way at any time. Therefore, a better model should not have the compromised nodes using almost exactly the same code as the uncompromised nodes, as our model does. Rather, it should allow compromised nodes to use specially written code that allows compromised nodes to change their behavior at any point in time and to collaborate with other compromised nodes. The model of a compromised node that we used when testing the DTM was limited to having only the DTM function incorrectly. However, any or all of the components of *S.A.F.E.* (and in any combination) should be able to function incorrectly in an accurate model. Simply put, a more accurate model of a compromised node should include the ability to do anything to fool uncompromised nodes and cause problems in a Trust Domain.

We suggest that when a more accurate model is developed, the developer should bear in mind that the goal of a compromised computer may not be just to destroy a Trust Domain. Instead, it may be trying to fool uncompromised computers into letting it access their resources. In this case, compromised computers may attempt to let uncompromised computers into the Trust Domain, as long as the compromised computers still will outnumber the uncompromised computers. In this instance, the compromised computers will be trying to keep the Trust Domain intact as long as possible to have the greatest opportunity to exploit their access to the uncompromised computers.

	Follower Behaves Normally	Forwards No Messages	Forwards Some Messages	Forwards Messages to Some Hosts
Leader Behaves Normally	N/A (Uncompromised Behavior)			
Leader Sends No Messages				
Leader Sends All Hosts Wrong Messages				
Leader Sends Some Hosts Wrong Messages				
Leader Sends Some Hosts Right Messages				
Leader Sends Some Hosts Wrong & Right Messages				
Leader Sends All Hosts Wrong & Right Messages				

Table 5 - The Possible Behaviors of a Compromised DTM in the Signed Message Algorithm

The model of a compromised node for the tests of *S.A.F.E.* was also a very simplistic one, and thus, future work should include studying it and developing it properly. We needed to use a test harness to test *S.A.F.E.* so that we did not have to compromise the systems that *S.A.F.E.* was running on. However, the test harness forced us to define the behavior of a compromised node. In testing the DTM, we had showed that a node with a compromised DTM would be detected. In testing S.A.F.E., we needed to verify that a node whose SHM has become compromised would be detected. Thus, while a node that was actually compromised might have a compromised SHM or DTM (or both), to perform the tests that would show whether S.A.F.E. worked correctly, we needed to have a compromised SHM. In our test harness, a compromised SHM would process every signature as an uncompromised SHM would, but it would return a different probability to the DTM than an uncompromised SHM would. For example, if an uncompromised SHM would return .5 for a signature, a compromised SHM would return some number not equal to .5 for the same signature. Although the model is simple, the compromised behavior selected for the SHM was carefully chosen from the set of possible behaviors. It is important to note that a compromised node in a Trust Domain will almost always be detected and removed if the node is accused by the uncompromised nodes. Thus, a compromised node must attempt to avoid being accused. When the compromised SHM, which has not yet been accused, receives a set of events that constitute the signature of a potential attack, it must return a probability that an attack has occurred to the DTM with which it communicates. If it returns a value that indicates it was definitely compromised or may have been compromised, then it sends a message requesting help, which results in the consensus algorithm being invoked. Then the computers in the Trust Domain will retest the node and discover it has been compromised. For example, an attacker that is trying to remain undetected will return only probabilities below the threshold where it is deemed that the node may have been compromised. We give the following example to help clarify.

In this context, assume computer C_1 is a compromised node in Trust Domain TD_a and C_1 conforms to the model of a compromised node that we have specified. Assume that the low threshold for a signature is .2 and the high threshold for a signature is .8. This means that any signature that causes an uncompromised SHM to return a probability below .2 is not an attack. Any signature which causes an uncompromised SHM to return a probability greater than or equal to .8 is an attack. Any signature which causes an uncompromised SHM to return a probability greater than or equal to .2 and less than .8 may be an attack. C_1 receives a signature that may be an attack with probability .5. If C_1's SHM sends a response of .5 to the DTM it communicates with, that DTM will instruct the other DTMs in TD_a to retest C_1. They will discover that C_1 compromised and remove it from the Trust Domain. If C_1's SHM sends a response of .9 to the DTM it communicates with, the DTM will inform the other DTMs in the Trust Domain that C_1 is compromised and they will remove it from the Trust Domain. Thus, C_1's SHM sends a response of .1 to the DTM it communicates with. That DTM believes that the SHM is not compromised and does not alert the DTMs on the other computers in the Trust Domain. Thus, C_1 has avoided being removed from the Trust Domain. It should be noted, however, that an attacker will almost always be caught eventually, if it returns an incorrect probability when it is retested.

4.2.1 Analysis Of Testing The DTM

In this first implementation of the DTM architecture and concepts, the functionality of the DTM was designed to accomplish the following four tasks correctly:

1. Allow uncompromised hosts to join a Trust Domain.
2. Keeps compromised hosts from joining a Trust Domain.
3. Detect and remove hosts that have become compromised after joining a Trust Domain.
4. Prevent a class of Denial of Service attacks where a compromised node continuously attempts to join and Trust Domain.

The results in table 1 indicate how well the DTM performed these tasks. There are two distinct sets of results in Table 1. The first set is the results obtained when there was a majority of uncompromised computers in the Trust Domain (shaded blue in Table 1), which is a fundamental assumption we stated in section 2.3.2. These results are consistent with our expectations and indicate that the DTM performed correctly for the four tasks. These results are also the crucial set, as they are relevant in support of our hypothesis. The DTM performed perfectly during the testing when the Trust Domain contained a majority of uncompromised nodes, leading us to believe that the Byzantine Agreement Protocol can be successfully applied to the computer security field in the area of intrusion detection and countermeasures systems.

The second set of results was obtained when there was a majority of compromised computers in the Trust Domain (shaded red in Table 1). These results indicate what may occur if our fundamental assumption in section 2.3.2, that the majority of nodes in a Trust Domain are not compromised, is violated. We believed that if there is a majority of compromised hosts in a Trust Domain, it has been taken over by compromised computers and may no longer function correctly. The second set of results does satisfy the preceding claim, and thus, these results were consistent with our expectations.

If the majority of hosts in a Trust Domain were compromised, then our claim that the Trust Domain might not function correctly would imply that

1. Uncompromised hosts **might** not be allowed to join a Trust Domain.
2. Compromised hosts **might** be allowed to join a Trust Domain.
3. Hosts that become compromised after joining a Trust Domain **might** not be detected and removed.
4. Repeated attempts to join a Trust Domain **might** not be prevented.

It is important to note that a Trust Domain may function correctly even if there are more compromised nodes than uncompromised ones, and thus, there were successes in some trials. This is due to the restricted model of a compromised node, discussed in the beginning of this section. However, if model of a compromised node had allowed the compromised nodes to collaborate intelligently, which may have included communicating with other compromised nodes, they could consistently prevent a Trust Domain from functioning correctly.

In our experiments, we also considered the case where the majority of the nodes in a Trust Domain were compromised, as shown in the red portions of Table 1. In this case, the experimental data did not support our original hypothesis, which was "if the majority of nodes in a Trust Domain are compromised, then, an uncompromised node should be rejected from the Trusted Domain while a compromised node should be allowed to join." At first, this seems to be an irreconcilable result. However, upon further reasoning it becomes clear that the results are consistent with the experimental design. A key element of the experimental design is the definition of compromised node behaviors. In our limited experimentation, the behavior of a compromised node was restrictive, in the sense that it simply provided faulty answers (i.e. the SHM provided wrong probabilities) or good answers statically. In addition, a compromised node was defined in such a way that SHM was the only misbehaving component, and not DTM. Clearly this is a very restrictive model of an attacker. On the other hand, if the attacker's model incorporates behavior such as "a compromised node will allow all requesting nodes in", then compromised nodes will have become members of a Trusted domain in all cases. The key lesson that this limited experimentation provided was to highlight the importance of clearly articulating and modeling the attackers' behaviors. Such modeling needs to be incorporated as a major element of this work, but it is clearly beyond the scope of this Thesis. For more information about this, the reader is encouraged to refer to recent works that address the problem of modeling attackers.

We expected that the one type of denial of service attack that the DTM can prevent would not be prevented by a Trust Domain with a majority of computers that are compromised. However, this expectation was not met. The key reason here is that the model of a compromised node is too simplistic. Even when a node is compromised it continuous to execute the denial of service attack prevention strategy unchanged, and therefore, it will still work correctly.

The performance results in table 2 show that the DTM used a small amount of the available RAM on the computer. The most memory that the DTM ever used was just less than 22 MB, and thus just less than 17 % of the 128 MB of installed RAM. The DTM also used only 7 % of the CPU time on the computer, on average, while it was running. This seems like an acceptable value compared to other well known Intrusion Detection Systems, like Bro, which can consume a gigabyte of RAM and cause average CPU utilization values of 60% [DR 04].

It would have been desirable to do significantly more testing. Comprehensive testing of the system would include testing all the features of the DTM with all possible numbers of computers in a Trust Domain (3, 4, 5, ..., 16). It would have also have included a more significant number of trials for each test. All the different combinations of behaviors of compromised nodes should be tested. In addition to this, the DTM should be tested with compromised nodes that are specifically designed to prevent the DTM from functioning correctly. The testing was all done manually, which is very time consuming. In the future, an automated testing solution, such as *Expect* (http://expect.nist.gov), should be used. An automated testing tool would allow significantly more tests to be run in less time, thus providing results that are much more statistically significant. It would also ensure that tests are reproducible. Automated testing should also be used to test *S.A.F.E.*

4.2.2 Analysis Of Testing *S.A.F.E.*

4.2.2.1 Results

When Snort detects an event that matches one it is configured to detect, it outputs a number corresponding to the perceived threat level of the event. The value is 1, 2, or 3. 1 corresponded to the highest threat level and 3 corresponded to the lowest threat level. Thus, when a computer Snort was monitoring was attacked, Snort outputted a value for each event in the signature. Because this is how Snort works, it only gives an indication of whether an attack has occurred, and does not indicate whether the attack has been successful. Therefore, a human must monitor the output of Snort and determine whether an attack has been successful. This takes time and a human can make a mistake. In addition, because Snort is monitoring network traffic, it can generate limitless output, which a human must then analyze.

Every time a computer in the Trust Domain was attacked, rather than just outputting a number to a log file like *Snort* does, *S.A.F.E.* successfully determined whether the node had been compromised by the attack. If the computer had been compromised, *S.A.F.E.* removed it from the Trust Domain. This was done without human intervention. Because *S.A.F.E.* interprets the values that Snort outputs and automatically tests a node that is attacked, it does not require a human to monitor it.

4.2.2.2 Resource Utilization

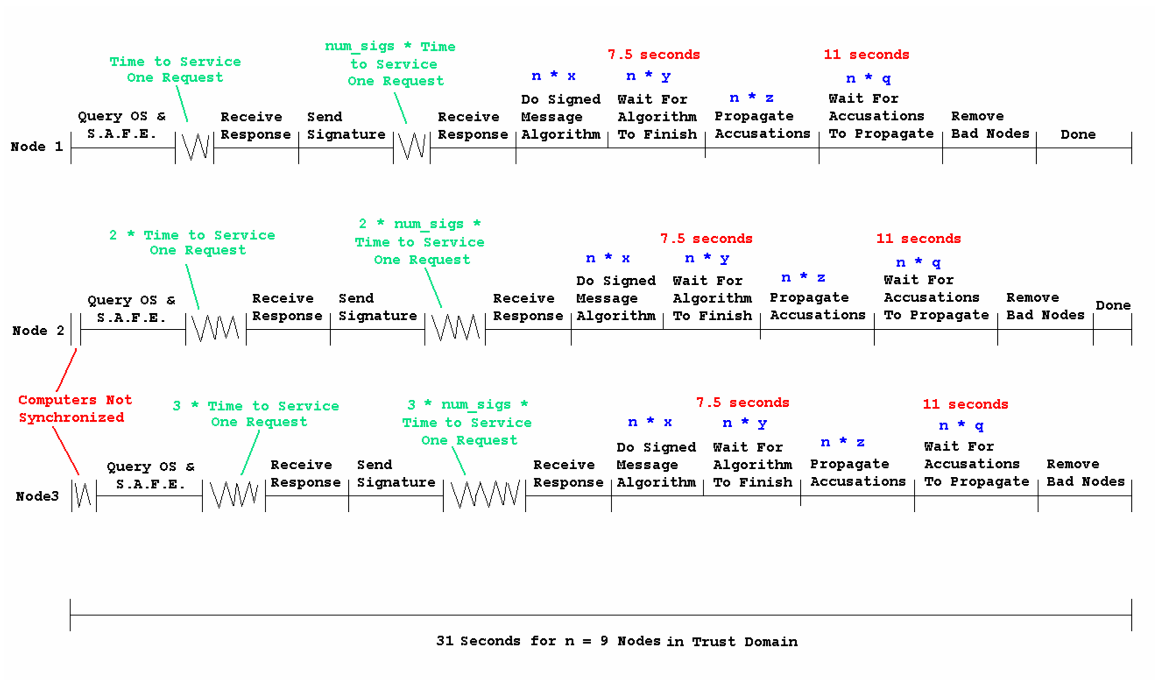
While *S.A.F.E.* was running, our measurements using *top* indicated it used less than 10% of the CPU time on each computer and used about 22 MB of RAM. In one trial, SAFE appeared to use about 32 MB of RAM. However, this occurred only in a trial that was run immediately after the computers were rebooted. More trials were performed and they all produced results consistent with all the other trials. Finally, the computers were all rebooted again, and one more trial was run. In this case, again, *S.A.F.E.* appeared to use about 32 MB of RAM. Running one final trial and examining the status file in each directory for the three processes that *S.A.F.E.* uses indicated that the resident set size of the memory for *S.A.F.E.* actually was about 32 MB. We note that there was a significant amount of unused memory (more than 30 MB), so we are confident that all of *S.A.F.E.*'s memory consumption was indicated by the resident set size. This leads us to believe that *S.A.F.E.* really does use about 32 MB of RAM. This also suggests that our measurements for the memory consumed by *S.A.F.E.* in section 4.2.1 also were incorrect and should have been approximately 32 MB of RAM. We suggest a more accurate tool be used to measure the exact amount of memory used. However, we are confident that the 32 MB is an accurate estimate of the memory used by *S.A.F.E.* We believe our measurements were inaccurate because *top* only gives the memory usage of the 20 processes consuming the most CPU time. Further analysis of the output showed that only two of *S.A.F.E.*'s three processes were being shown by *top*. We point out that more memory may be consumed by *S.A.F.E.* when the maximum number of nodes (16) join a Trust Domain. Extrapolating from the numbers in the resident set size of the DTM process, we expect that no more than 10 MB of additional memory would be required for this situation. This is because the DTM appears to use 10 MB of memory with 9 nodes in the Trust Domain, and the memory usage of the DTM scales linearly with the number of nodes in a Trust Domain, due to the way it is implemented.

4.2.2.3 Scalability Analysis

One of the requirements for *S.A.F.E.* is that it must be scalable. In this research, we focused on the scalability within a Trust Domain. We established that a Trust Domain should work with 16 nodes in section 2.3.3, and we discuss the scalability of *S.A.F.E.* to thousands of nodes in section 5.2, based on the assumption of 16 nodes per Trust Domain. In order to conclude that a Trust Domain is scalable to 16 nodes, we needed to show that *S.A.F.E.* functioned efficiently enough to be a potentially viable IDCS when there were 16 nodes in a Trust Domain. This means that the DTM must be able to efficiently determine whether a node will or will not be admitted into a Trust Domain. It also requires that the nodes in a Trust Domain be able to, in an acceptable amount of time, retest any node that may have become compromised. The definition of "acceptable" should be based on information in the fields of cognitive science and computer security. This information should include how long a wait that users will tolerate and the rate at which a compromised node will damage the other nodes in a Trust Domain. While we do not have formal definitions for what values are acceptable, we believe that for our purposes, 60 seconds and 30 seconds are acceptable amounts of time, respectively in the context of *S.A.F.E.* We note that the algorithm to retest a node is the same as the algorithm to determine whether a node should be admitted into a Trust Domain, and for simplicity, we will refer to it as $\text{Algorithm}_{\text{test}}$ henceforth.

Our first measurements of how long $\text{Algorithm}_{\text{test}}$ took to run were so poor, that we were only able to run the tests for a Trust Domain with nine nodes. The average time it took to run $\text{Algorithm}_{\text{test}}$ was 174 seconds. One factor that contributed to this was that the nodes in a Trust Domain must be synchronized at several points during $\text{Algorithm}_{\text{test}}$. In order to avoid the synchronization problem during development of the DTM, we had introduced code that forced the DTM to sleep at two points during the $\text{Algorithm}_{\text{test}}$. During our first measurements, these times were 60 seconds and 90 seconds, respectively. This allowed all the nodes in the Trust Domain to stay synchronized enough for the algorithm to work correctly. Eventually we concluded that these delays were not caused by the DTM, and that there must be a problem with how the SHM was implemented, causing it to take significantly longer to process a signature than it should. This delay caused the need for long sleep times in the DTM. We modified the SHM to remove the performance impediments, decreased the times that the DTM sleeps, and then measured how long it took for $\text{Algorithm}_{\text{test}}$ to run.

After reducing the sleep times from 60 seconds to 7.5 seconds and from 90 seconds to 11 seconds, $\text{Algorithm}_{\text{test}}$ only took 31 seconds with nine nodes in the Trust Domain. Chart 2 shows a high-level view of how the DTM used the time to run $\text{Algorithm}_{\text{test}}$. We believe that we can reduce the sleep times even more and still have $\text{Algorithm}_{\text{test}}$ function correctly.



Node 1
Query OS & S.A.F.E.
Time to Service One Request
Receive Response
Send Signature
num_sigs * Time to Service One Request
Receive Response
n * x
Do Signed Message Algorithm
7.5 seconds
n * y
Wait For Algorithm To Finish
n * z
Propagate Accusations
11 seconds
n * q
Wait For Accusations To Propagate
Remove Bad Nodes
Done
Node 2
2 * Time to Service One Request
2 * num_sigs * Time to Service One Request
Node3
Computers Not Synchronized
3 * Time to Service One Request
3 * num_sigs * Time to Service One Request
31 Seconds for n = 9 Nodes in Trust Domain

Chart 2 – How the DTM's Time is Spent

However, it should be noted that in our proof-of-concept implementation, three features are not implemented. These features will increase the amount of time that it takes the DTM to determine if a host is compromised. The first feature that is not implemented is the signing (and verification of signatures) of messages. According to professor Berk Sunar of WPI, signing and verifying the signature of a message should take only milliseconds on an Intel® Pentium® 4 processor, and therefore, this will not constitute a factor of any importance in the implementation of the algorithm. The second feature is ensuring the node being tested is running software that is acceptably patched, including an operating system. This event only occurs when a node attempts to join a Trust Domain, and therefore it should not be a significant factor in the dynamic nature of the algorithm and/or affect scalability.

The third feature is sending multiple signatures, instead of just one, to determine if a host is compromised. This feature will increase the time it takes to determine whether a host is compromised. This is because all the signatures must be processed by the SHM, so the SHM becomes a bottleneck for performance, as it functions as a queue serviced by a single server. The increased number of signatures it must process will cause the DTMs on different hosts to get further out of synchronization than they already are, necessitating a new way to determine when all hosts are ready or a high sleep time so all hosts are ready to proceed to the next step. However, we are unsure how much additional time this will add. This is the critical factor. However, it is factor that could be managed by using design trade-offs. For example, the larger number of signatures and classes of signatures that are used, the more accurate the determination as to the trustworthiness of a node will be. Therefore, if you were to statistically measure the accurate assessment vs. number of signatures, and define a confidence interval, then, SAFE should be able to manage the scalability nicely.

Although the time required to perform the Signed Message Algorithm and propagate accusations will increase as the number of nodes in the Trust Domain increases to 16, we do not believe that this would have as large an impact on the time it takes to determine whether a host is compromised. The queuing problems will outweigh this. Future work should include running tests to determine exactly how much time it would take to determine whether a node is compromised if there are 16 nodes in a Trust Domain.

We also note that *S.A.F.E.* was tested on a system running nothing else. Running a benchmark on the system at the same time as *S.A.F.E.* would be a much more realistic test. We did not do this because the computers we had were so old that it was clear this would cause even more problems. However, since the DTM must send O(n^3) messages over the network when it tests a node, a benchmark that put traffic on the network card would have been a good choice, because the DTM is neither CPU bound nor disk I/O bound, but rather network bound. The network traffic of the benchmark may cause delays in the sending and receiving of messages, which will negatively affect *S.A.F.E.*'s performance. This would cause the nodes to get further out of synchronization with each other, and it could cause good nodes to be removed from the system because they may not respond quickly enough to messages.

Even with the additional time that will be required for a node being tested to perform the tasks that have not been implemented yet, we believe that the number of nodes in a Trust Domain will scale to 16. We believe that the additional computations that must be performed will add very little processing time. At the same time, over half of the time to determine if a node is compromised is currently spent sleeping, and we believe that we can reduce that significantly. We believe that there are two ways we can do this. The first method is to reduce the sleep times as low as they can go, to just before the point where they would cause the DTM to malfunction. We also could implement another round of messaging designed to complement the sleep times. Each node could send a message to all the other nodes when it is ready to proceed. When all the nodes have reported in, all the nodes could stop sleeping and continue to run the algorithm. The sleep times could then be used as a maximum amount of time to sleep, in the event that some node does not send its "ready message."

5 Conclusions And Future Work

5.1 Conclusions

The WPI System Security Research Lab's *S.A.F.E.* project attempts to design an intrusion detection and countermeasure system that addresses the problems that all IDCSs we know of suffer from. Previous work addressed the problems of feature selection and pattern categorization and adaptation. In this work, we have devised a way to attack the problem of fault tolerance and resistance to subversion. We have designed and created an implementation that integrates seamlessly into the existing framework of *S.A.F.E.* Our test results lead us to believe that the Byzantine Agreement Protocol can successfully be applied to the domain of computer security in the area of intrusion detection and countermeasure systems. However, the implementation is far from complete, as the future work section will discuss. There are several features that need to be implemented and a great deal of performance tuning to be done. In addition to this, our proof-of-concept implementation has not been tested thoroughly enough to conclude that it is a success. The tests merely indicate that it is a good idea conceptually and point out some potential problems. Our earlier concerns about the scalability of our implementation were due to faulty experimental data, and we do not believe that they are well-founded. Indeed, the results described in section 4 suggest, as a first order approximation, that scalability will not be a problem. Having said this, we still need to complete a significant set of tasks to unequivocally confirm that scalability is not an issue. These tasks include: (1) define, design, and implement a more realistic model of an attacker; (2) design and implement all the missing features of the DTM mentioned in sections 4 and 5.2; (3) perform a more robust and complete set of functional, performance, and scalability tests. This testing should be done once the performance tuning suggested in the previous section is completed.

5.2 Future Work

We have implemented the portion of a basic framework for a distributed intrusion detection and countermeasures system that deals with fault tolerance and resistance to subversion, based on a variant of a solution to the Byzantine Generals Problem. However, our implementation is based on many assumptions that we have made. Also, significant portions of the system have not been implemented yet. Future work should address these assumptions and complete the implementation of the DTM. The remaining tasks range from mundane implementations using existing methods to more innovative ideas, which require further research. The future work can be classified under several different problems, such as masquerading, the distribution and protection of sensitive data, scalability, and implementation issues.

The current implementation of the DTM does not address the problem of masquerading, where a node intentionally misrepresents itself. Two examples of the masquerading problem are compromised nodes pretending to be other nodes and groups of compromised nodes pretending to be Trust Domains. For example, a compromised host might try to join the Trust Domain repeatedly, using the IP Address of another host that might want to join the Trust Domain. After some number of unsuccessful attempts, the nodes in the Trust Domain would deem that the attacking node was trying to cause a denial of service attack. All subsequent requests from its spoofed IP Address would be ignored for a period of time. If the host that legitimately was using that IP Address then tries to join the Trust Domain, its requests would be ignored for a period of time, causing a true denial of service. A single host or a group of compromised hosts may listen to the multicast channel for a legitimate Trust Domain and convince a host trying to join that they are part of the Trust Domain the host is trying to join. If they fool the intended victim, this could cause a Trust Domain to get out of synchronization.

The problem of distribution and protection of sensitive data is crucial to *S.A.F.E.*'s success. This work has assumed the existence of a method that prevents compromised computers from obtaining the data used to test hosts. Although we gave some thought to this issue, we did not reach a conclusion on how it should work. Instead, we present two possible ways to do this, and our reasoning. The data cannot be stored in a central repository. If it is, then if the host containing that repository became compromised, all the data would become useless, because compromised hosts would have access to it. One way to prevent compromised computers from obtaining the data is to generate the data as it is needed and delete it immediately after it has been used. In order to generate the data dynamically, each host would have to have a method of generating the data such that no duplicate signatures are created by multiple nodes. This could be done using some unique host properties such as the MAC address of the host's Ethernet card or IP address. If uncompromised hosts generate the signatures and corresponding probabilities as they need them and delete them immediately after using them, then no uncompromised host will have the data. Thus, compromising a host will not yield any data. Another method is to partition the data such that each host in a Trust Domain only has a subset of it. Thus, if a host is compromised, there is still more data that compromised nodes would not know. That data can be used to test new hosts. Shamir's secret sharing techniques could be used to do this [SH 79].

It is critical that *S.A.F.E.* is scalable enough to allow large numbers of computers to trust each other. We limited the number of hosts in a Trust Domain to 16, in order to keep the number of messages that need to be sent and thus the time required to decide whether a host should be allowed to join a Trust Domain, reasonable. In order to allow many computers to use S.A.F.E, hundreds or thousands of hosts must be able to communicate. We can address this problem using a hierarchical method, similar to the techniques used in networking. We can create many Trust Domains with 16 hosts. Each of these Trust Domains would then elect a leader. The task of leader must be rotated at various intervals, in case a leader becomes compromised or fails. The leaders of 16 Trust Domains would form another Trust Domain at the second level of the hierarchy. This Trust Domain would also have a leader, which would represent it in a Trust Domain at the next level of the hierarchy, and so on. In this manner, *S.A.F.E.* can allow 4096 hosts to work together. 4096 hosts at level 0 form 256 Trust Domains at level 1. The leaders of these Trust Domains form 16 Trust Domains at level 2, and their leaders in turn form 1 Trust Domain at level 3. To determine whether a host is trustworthy, a message can be passed up the hierarchy and back down to the leader representing the host in question.

That leader can then pass a message back through the hierarchy to the original sender with the correct information. However, in order for the leaders of Trust Domains to be able to form higher level Trust Domains, they will need a way to send their requests to join these higher level Trust Domains across subnet boundaries. In addition to this, hosts trying to join a Trust Domain at a level higher than level 1 will need to be able to discover the Trust Domain they are trying to join, or have a well-known location of that Trust Domain. If the location is well known, it will have to be given to the program as a command line argument or hard coded in the program. A well-known location is easily attacked and disabled by a denial of service attack, so the location needs to be discovered at runtime or passed as an argument. Because passing the argument may require human intervention in what we would like to be an automated process, it is desirable to have the location discovered at run time. This may be accomplished using overlay networks.

Finally, several implementation issues need to be addressed in the future. One task is to implement signed messages using well-known cryptography techniques. This will involve using public key cryptography to encode and decode each message sent by the parallel version of the Signed Message Algorithm that the DTM uses. This will ensure that the hosts be able to detect if a message was tampered with. Another task that needs to be completed is to query the operating system and installed software on the host trying to join. We believe that a method similar to those used by Windows Update or Linux's update program can be used to accomplish this. A shortcoming of the current implementation is that hosts must be in the same subnet to join a Trust Domain at level 1. This is because multicasting, as well as broadcasting, can only occur within a subnet. Thus, because hosts currently use multicast as the way to send their request to join a Trust Domain, only hosts in the same subnet as the host trying to join can receive the message. A timer-based mechanism to retest all the hosts in a Trust Domain at a configurable interval should also be implemented. The only concern is how to prevent multiple requests to retest the hosts based on the timer. We believe that this could be addressed using the same method we use to prevent the one specific type of denial of service attack that DTM prevents. Finally, the Trust Matrix should be expanded for different host and network based services, as well as subjects within a node. This would allow nodes to communicate with certain services on other nodes, if they felt the services were trustworthy. At the same time, they could choose not to communicate with services they felt were untrustworthy.

Acknowledgement:

This work was supported in part by a grant from the Acumen Consulting Group, Marlboro, Massachusetts.

7 References

[AC 99] Julia Allen, Allen Christie, William. Fithen, John McHugh, Jed Pickel, and Ed Stoner, “State of the Practice of Intrusion Detection Technologies,” CMU/SEI, Pittsburgh, PA, Tech. Rep.-028, 1999. [Online]. Available: http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr028.pdf

[AX 00] Stefan Axelsson “Intrusion Detection Systems: A Taxonomy and Survey,” Dept. of Computer Engineering, Chalmers University of Technology, Sweden, Tech. Rep. No 99-15, March 2000 [Online]. Available: http://citeseer.ist.psu.edu/cache/papers/cs/13832/http:zSzzSzwww.ce.chalmers.sezSzstaff zSzsaxzSztaxonomy.pdf/axelsson03intrusion.pdf

[BA 98] Jai Balasubramaniyan, Jose Omar Garcia-Fernandez, Eugene Spafford, and Diego Zamboni, “An Architecture for Intrusion Detection using Autonomous Agents,” Department of Computer Science, Purdue University, Coast TR 98-05, 1998. [Online]. Available: https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/98-05.pdf

[BI 94] Biswanah Mukherjee, Todd L. Heberlein, and Karl N. Levitt. “Network Intrusion Detection,” *IEEE Network*, vol. 8(3), pp. 26-41, May/June 1994. [Online]. Available: http:\\www.ieee.org

[BI 04] Bro Intrusion Detection System – Bro FAQ. [Online]. Available” http://www.bro-ids.org/FAQ.html

[BU 02] Pedro Bueno. “*Paranoid Penguin: Understanding IDS for Linux*,” *Linux Journal*, vol. 2002 Issue 97, May 2002. [Online]. Available: http://www.linuxjournal.com/issue/97

[CO 02] Fernando C. Colon Osorio, “An Overview of Intrusion Detection & Countermeasure Systems – Research Directions.” [Online]. Available: http://www.cs.wpi.edu/~fcco/IDCS-10212002.ppt

[DE 87] Dorothy Denning, “An Intrusion Detection Model,” *IEEE Transactions on Software Engineering*, vol. 13 (2), pp. 222-232, Feb. 1987.

[DR 04] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer, “Operational Experience with High-Volume Network Intrusion Detection,” in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004, pp. 2-11.

[GS 91] Simson Garfinkel and Gene Spafford, *Practical Unix Security*. Sebastopol, California: O'Reilly and Associates, 1991.

[HA 01] Haines, Joshua W., Lee M. Rossey, Richard P. Lippmann, and Robert K. Cunningham, “Extending the DARPA Off-Line Intrusion Detection Evaluations,” in

Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEXP II), 2001. [Online]. Available: http://www.ll.mit.edu/IST/pubs/discex-jhaines-2001-final.pdf

[HM 00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Boston: Addison Wesley, 2001.

[LA 96] Butler W. Lampson, "How to Build a Highly Available System Using Consensus" in *Lecture Notes in Computer Science* vol. 1151, 1996, pp. 1-17. [Online]. Available: http://research.microsoft.com/lampson/58-Consensus/Acrobat.pdf

[LI 99] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das, "*The 1999 DARPA Off-Line Detection Evaluation*".

[LL 82] Leslie Lamport, Robert Shostak, and Marshall Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4(3), pp. 382-401, July 1982. [Online]. Available: http://portal.acm.org/dl.cfm

[KO 03] Jack Koziol, *Intrusion Detection with Snort*. Indiana: SAMS Publishing, 2003.

[PA 99] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time", *Computer Networks*, vol. 31(23-24), pp. 2435-2463, Dec. 1999. [Online]. Available: http://www.icir.org/vern/papers/bro-CN99.html

[SH 79] Adi Shamir, "How to Share a Secret," *Communications of the ACM* vol. 22 (11), 1979, pp. 612-613. [Online]. Available: http://portal.acm.org/citation.cfm?id=359176&coll=portal&dl=ACM

[SN 91] William E. Snaman, Jr., "Application Design in a VAXcluster System" *Digital Technical Journal of Digital Equipment Corporation* vol. 3 (3), 1991, pp. 16-26. [Online]. Available: http://www.hpl.hp.com/hpjournal/dtj/vol3num3/vol3num3art2.pdf

[SP 00] Eugene H. Spafford and Diego Zamboni, "Intrusion Detection Using Autonomous Agents", *Computer Networks*, vol. 34, pp. 547-570, Oct. 2000. [Online]. Available: http:\\www.sciencedirect.com

[TA 04] Talisker Security Wizard. [Online]. Available: http://www.networkintrusion.co.uk/N_ids.htm

[TW 04] Tripwire Open Source, Linux Edition FAQ. [Online]. Available: http://www.tripwire.org/qanda/index.php

Appendix A - Functional Specification

DTM Functional Specification

1. This Document:

   This document describes the functional requirements for the Distributed Trust Manager (DTM). In addition to functional requirements, this document lists assumptions that must be made in order to implement DTM in a reasonable amount of time. These assumptions might be good to address as part of a dissertation.

2. Functional Requirements:

   Version 1.0 (Masters Thesis)

   - DTM must be able to communicate with SHM using a well defined protocol.
   - All members of a Trust Domain will query a new system that requests to join the Trust Domain.
   - A new system will be allowed to join a Trust Domain, if and only if all members of a Trust Domain agree it should be allowed to join.
   - The systems in a Trust Domain will detect nodes in the Trust Domain that have been compromised and remove them from the Trust Domain.
   - DTM will run on any computer with an operating system supporting Java 1.4.1 or better.

   Version 2.0

   - The first node brought up will create a Trust Domain and provide a mechanism for other nodes to locate the Trust Domain.
   - DTM will allow the creation of additional Trust Domains when the number of systems exceeds the allowed number for a trust domain.
   - Trust domains must be able to communicate with each other.
   - Nodes in a Trust Domain wishing to determine if a node in another Trust Domain will be able to determine whether the node is trustworthy by the leaders of the trusted domains communicating.
   - A mechanism to "elect" a leader must exist.
   - The task of being a leader will be rotated.
   - There will be a way for test data to be distributed amongst the nodes of a trust domain.

3. Assumptions:

   The following assumptions are made. These should be addressed in a future phase of the project, if it continues.

- We assume that all messages are signed. However, we do not implement the cryptographic algorithms required to actually sign the messages.
- We assume that DTM will work on any computer with an operating systems that support Java 1.x.y or better. However, we will only test it on the following operating systems:
    - Debian Linux
- We assume that the method of checking system requirements of a node trying to join a Trust Domain exists.
- We assume that the first 3 nodes that form a Trust Domain are uncompromised.
- We assume that the countermeasures that bring a system back to a good state after it has been compromised exist and work.

Appendix B - Architectural Specification

DTM Architecture Specification

1. This Document

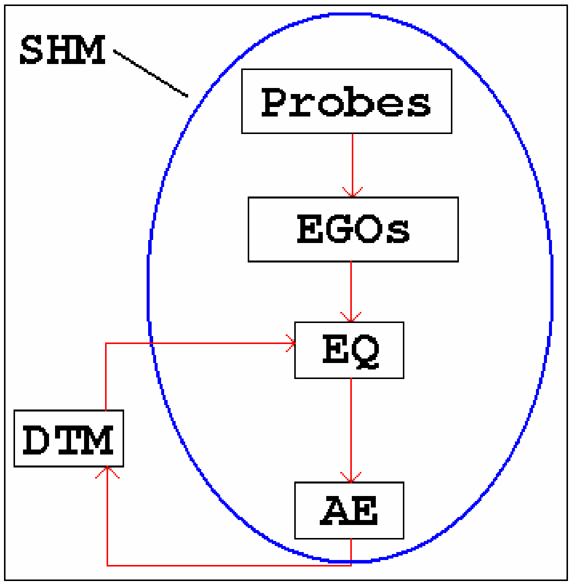
This document describes the architecture for the Distributed Trust Manager (DTM) and the reasoning for the architectural decisions. In addition to architectural decisions and the reasoning for them, this document lists assumptions that must be made in order to implement DTM in a reasonable amount of time. These assumptions might be good to address as part of a dissertation.

2. Architecture

A. *S.A.F.E.* Architecture

The *S.A.F.E.* architecture imposes some requirements on the way DTM works, and thus the DTM architecture. The *S.A.F.E.* architecture is described below, so that its impacts on the DTM architecture can be seen.

S.A.F.E. consists of 5 major components: probes, Event Generator Objects (EGOs), the Secure Host Manager (SHM), the Event Queue (EQ), and the Distributed Trust Manager (DTM). Each system with the *S.A.F.E.* software installed on it has 1 instance of each of these modules. The probes collect events that occur on the system and the EGOs convert those events into a common format and put them into the EQ. The DTM also can put data into the EQ. SHM retrieves events from the EQ and processes them. Based on the events the SHM retrieves, it may output a message to the DTM on the system.

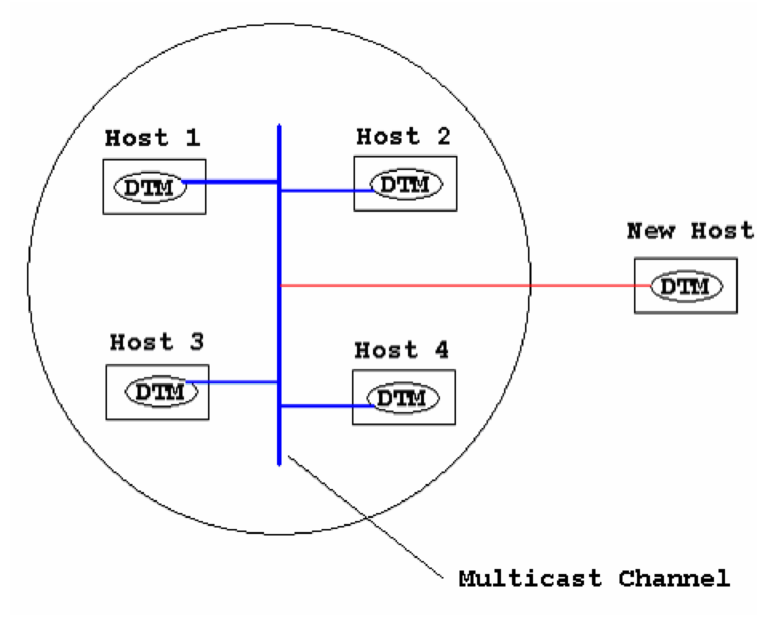


***S.A.F.E.* Architecture**

B. DTM Architecture

The DTM architecture specifications are listed in the order that they will come into play when the DTM is started.

A broadcast channel exists for each trust domain. Each node in a trust domain listens to the broadcast channel. When a node requests to join the trust domain, it will send a message in on the broadcast channel and all (uncompromised) nodes must react by querying the node. A mechanism must exist for the nodes to ignore repeated requests from a node such that if it floods the channel with requests, it does not impede normal work from being done. This means, however, that it could still cause a denial of service attack by sending multiple requests as fast as it can. If a multicast channel was not used, a leader would have to be contacted to initiate the allow/deny mechanism. A compromised leader could ignore requests from nodes to join the trust domain or cause the other nodes to keep querying a “potential node”, thus not getting any useful work done. This will be implemented using Java’s MulticastSocket class, which was designed for this purpose.

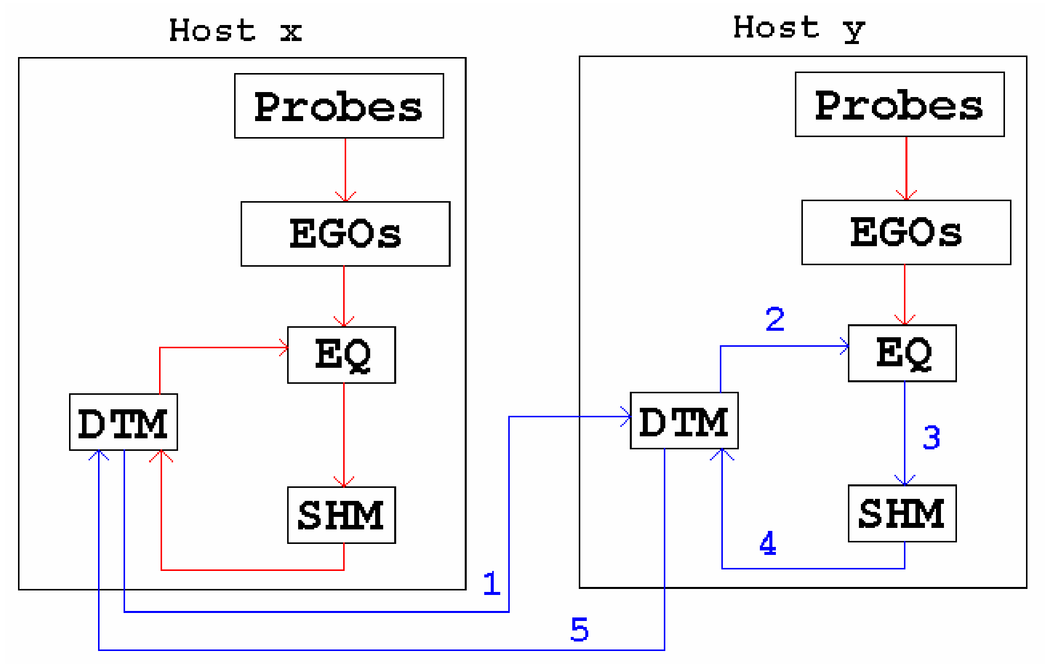


New Host Requests to Join Trust Domain

Instances of the DTM must talk to each other directly in order to implement Lamport et al.'s Signed Message Algorithm. This is accomplished by using sockets to simulate a direct connection.

The DTM must also listen to the SHM on the same host. The DTM establishes a socket connection to the SHM on the same host and communicates with the SHM using a predefined protocol that has been detailed in the SHM documentation.

The DTM must be able to put events into the Event Queue. The DTM establishes a socket connection to the Event Queue on the same host and communicates with the Event Queue using a predefined protocol that has been detailed in the Event Queue documentation.



Host x interrogates Host y

3. Assumptions:

The following assumptions are made. These should be addressed in a future phase of the project, if it continues.

- When the DTM is started on a host, it is given a parameter to indicate that it should create a trust domain or join an existing trust domain. If it is given a parameter instructing it to join an existing trust domain, then it is also given a parameter indicating which trust domain to join. This allows us to not use the wireless technique of searching for a trust domain the way a wireless device searches for a network when it's started. This way of doing things means that an administrator will need to initiate the creation of trust domains, and that trust domains cannot be automatically created. When a trust domain contains the maximum number of hosts, when a new host requests to join the trust domain, it will be told it cannot join. When this happens, the administrator will have to tell the DTM to create a new trust domain.

Appendix C - Design Specification

DTM Design Specification

1. **This Document:**

   This document describes the design for the Distributed Trust Manager (DTM). In addition to the design of the DTM, this document lists assumptions that must be made in order to implement DTM in a reasonable amount of time. These assumptions might be good to address as part of a dissertation.

2. **Design:**

   The design of the DTM is intended to be modular, so individual pieces can be easily replaced if needed, as the project matures and evolves. In order to accomplish this, the DTM was designed using standard object oriented design techniques. Design decisions were required to implement several of the functional requirements of the DTM. Those decisions and the reasoning behind them are described below. The appropriate design diagrams are shown after the rational for design decisions.

   <u>Requirements</u>

   Requirement: A host must be allowed to request to join a trust domain.

   Decision: When a trust domain is created, the host creating it will create a multicast channel using Java's MulticastSocket class. All requests to join a trust domain will be sent to the multicast channel.

   Rationale: In order to ensure that a host's request to join a trust domain is received by all members of a trust domain, the request must be sent to at least one uncompromised member of the trust domain. This is because compromised hosts might ignore the request and never respond to it, thus not allowing an uncompromised node to join a trust domain. To ensure that an uncompromised host receives the request, the request must be sent to multiple hosts in the trust domain. In order to do this, there must either be a well known list of hosts in the trust domain or a way to ensure that multiple hosts receive the request. Because trust is not a static thing, hosts will join and be removed from trust domains, and a well known list would have to be updated frequently. This list would also need to be stored in a well known location not on any host in a trust domain, since a host might be compromised and prevent proper access to the list. If the list is published on a host, it may be subject to attack. Thus, unless the list is specified as an input file, which requires significant user intervention and work, it is not a viable solution. We believe that the additional user intervention and work is too demanding to be useful since the list may have to be updated frequently. Thus we do not believe that a list is a viable option.

Another method of ensuring that multiple hosts in a trust domain receive a request is broadcasting the message to all hosts that might be in a trust domain. This can be accomplished using broadcast. Broadcasting the message sends it to all members of a subnet. This limits trust domains to having only hosts in one subnet, in order to ensure that an uncompromised host receives the request. It also sends the message to many other nodes that should not receive the message, using CPU time on hosts that do not need to receive the message to process it. Multicasting is similar to broadcasting, but allows the message to only go to hosts that have specifically requested the request. The processing is done at a lower level in the networking model, thus not consuming as much CPU time to process unwanted messages. Both broadcasting and multicasting are done using UDP, which means that some requests may not be received by all hosts in a trust domain, so a host may have to send more than one request to join. This is a shortcoming of using these methods, but it is outweighed by the problems of the well known list technique. Because multicasting is better suited to our needs than broadcasting, we have decided to use it.

Requirement: Hosts that become compromised after joining a trust domain must be detected.

Decision: Lamport et al.'s Signed Message Algorithm is run in parallel, with each host in a trust domain acting as the leader in one run of it.

Rationale: Running the Signed Message Algorithm as specified in the decision has a cost of $O(n^3)$ for one execution of the algorithm, where n is the number of hosts in the trust domain. This method allows us to detect compromised hosts within the trust domain, as described below, and it achieves consensus about whether to let a host join the trust domain. In addition to just achieving consensus which one execution of the Signed Message Algorithm can do at a cost of $O(n^2)$, the parallel version guarantees that compromised nodes are not admitted to a trust domain, which can happen if the decision is based on one execution of the Signed Message Algorithm. The guarantee is only valid as long as a simple majority of the hosts in the trust domain are not compromised. However, this premise is reasonable unless a majority of hosts in a trust domain can be compromised after joining, but before the next time they are tested occurs, which is extremely unlikely. We have been unable to come up with a less costly algorithm that achieves consensus that will not allow compromised hosts to join a trust domain and determine that hosts in a trust domain have been compromised.

Compromised hosts in a trust domain are detected by using the parallelized version of the Signed Message Algorithm, assuming that a simple majority of the hosts within the trust domain are not compromised. This is a result because there are only a limited number of possible events which can occur when the Signed Message Algorithm is run.

If the leader is not compromised, then after running the Signed Message Algorithm in parallel, all hosts that are not compromised know that the leader is not compromised, because the leader sent the correct conclusion that all hosts that are not compromised determined.

If the leader of an execution of the Signed Message Algorithm is compromised, then it can send:

- 0 messages to all uncompromised nodes - All uncompromised nodes assume node compromised or dead (same treatment as compromised).

- 1 message to only some uncompromised nodes – The uncompromised nodes detect there's a compromised node in the system and send a message to other nodes accusing the suspected compromised nodes. The accused nodes are then tested and determined to be compromised or uncompromised.

- 1 message to all uncompromised nodes - All uncompromised detect message is wrong and host is compromised if it contradicts the majority. If message does not contradict the majority, this cannot be detected yet unless the node sends a different message to at least 1 compromised node that forwards the message to at least 1 uncompromised node. Although this node should be removed, it is not currently causing damage, so it is not a critical problem.

- 2 (or more) different messages to some uncompromised nodes - All uncompromised nodes see contradictory instruction and know node is compromised.

Requirement: There must be a way to select a leader for a trust domain and rotate the task of being the leader.

Decision: Whenever a leader of a trust domain must be selected, each host in the trust domain will contribute a random numbers The number from each host will all be added and the remainder of the sum when it is divided by 255 will be used to select the IP address of the next leader. The host with the IP address closest to the remainder will be the next leader. Every time a leader becomes compromised and is removed from the trust domain, a new leader will be selected. At every n seconds where n is passed as a command line argument by the system administrator, the leader will initiate the selection of a new leader. If a leader does not initiate the selection of a new leader in the specified amount of time plus some small epsilon, they will be determined to be compromised and removed from the system and a new leader will be selected.

Rationale: This method of selecting a host is a low cost, random method for selecting a leader and ensures that the task of leader is rotated.

Requirement: DTM attack data must be distributed to prevent a compromised host from destroying the security of the entire trust domain.

Decision: When a host is allowed to join a trust domain, it will then generate the attack data that it will use to interrogate other hosts. The generation will be based on a seed that is passed as a command line argument when the DTM is started. If another host in the trust domain is using the same seed, a new one will be generated for the new host. This will be done randomly by the trust domain by each host submitting a random number and adding them all together. Collisions with other existing seeds will be resolved by the same method. Attack data will be randomly generated after each usage.

Rationale: This method means that no attack data is duplicated. It also does not require a central repository, so no one host can be compromised and destroy the security of the entire trust domain. Since the attack data is only used once, no compromised node can allow other compromised nodes to join the system by revealing the data to them.

Requirement: DTMs must communicate with other DTMs in the same trust domain.

Decision: All communication between DTMs in the same trust domain will take place over sockets.

Rationale: This method ensures that messages are received correctly because it uses TCP/IP.

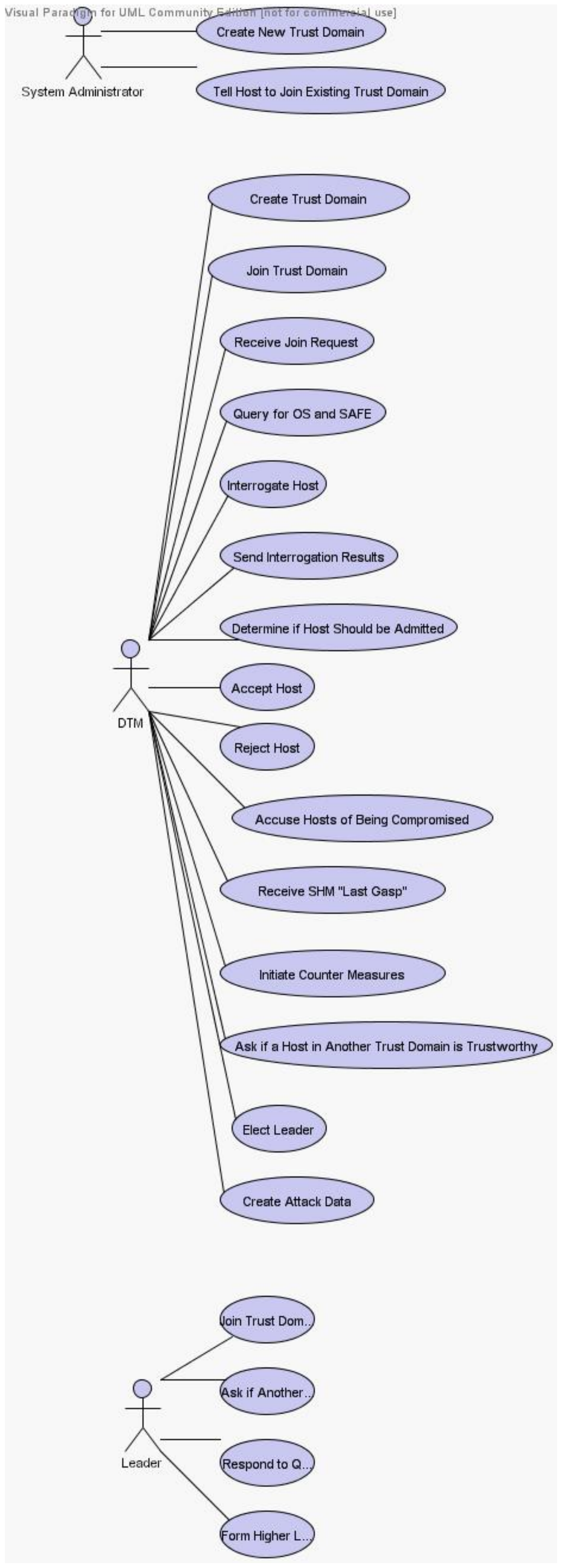
Requirement: There must be a way for leaders in different trust domains to communicate to inquire if a host in the leader's trust domain is trustworthy.

Decision: We will assume that the leaders of trust domains are well known. This information will be communicated off-line.

Rationale: In order to proceed with the design of the DTM, we must know generally how the leaders in different trust domains will communicate with each other. We could use a centralized location to keep track of the leaders, but this method is not viable. It allows a simple denial of service attack against the central location to render it impossible for leaders in different trust domains to communicate with each other. We could also communicate the information with a trusted protocol that we could design, but that is beyond the scope of the first version of the DTM. The other way trust domains could communicate with each other is using an overlay network. Using overlay networks would allow the flexibility we need to have changing leaders, but implementing it is beyond the scope of this project, and will be left for the next version of the DTM.

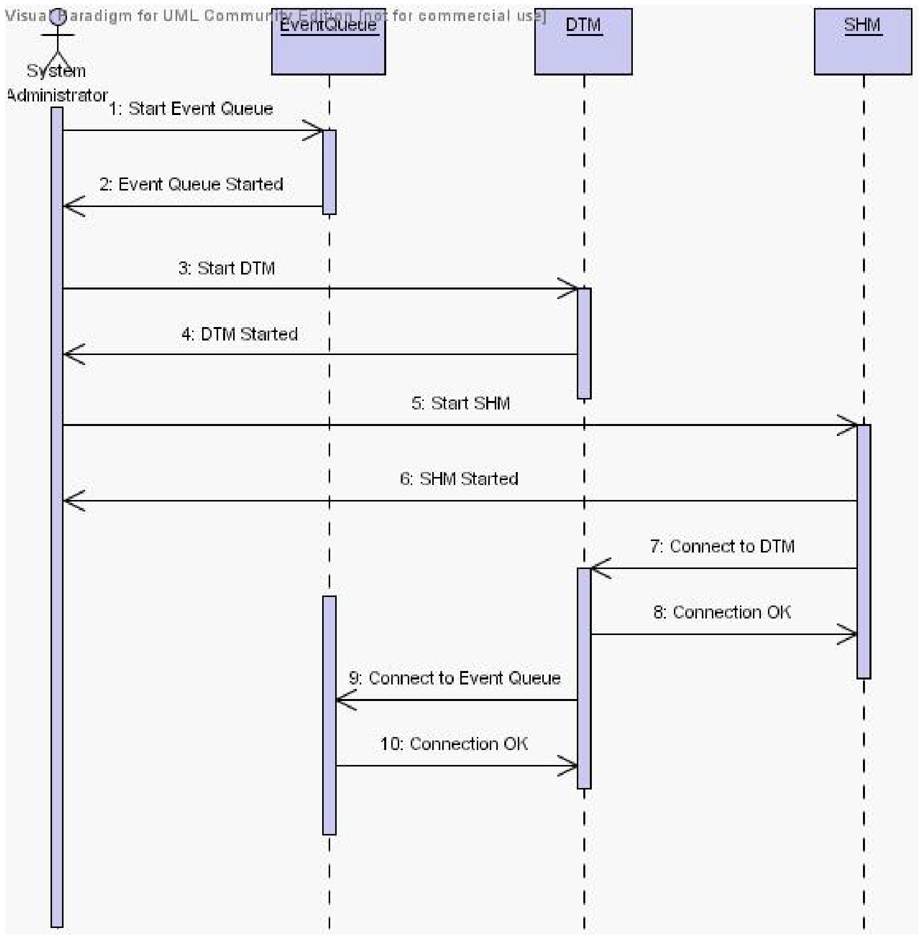
Object Oriented Design Diagrams

Use Case Diagrams

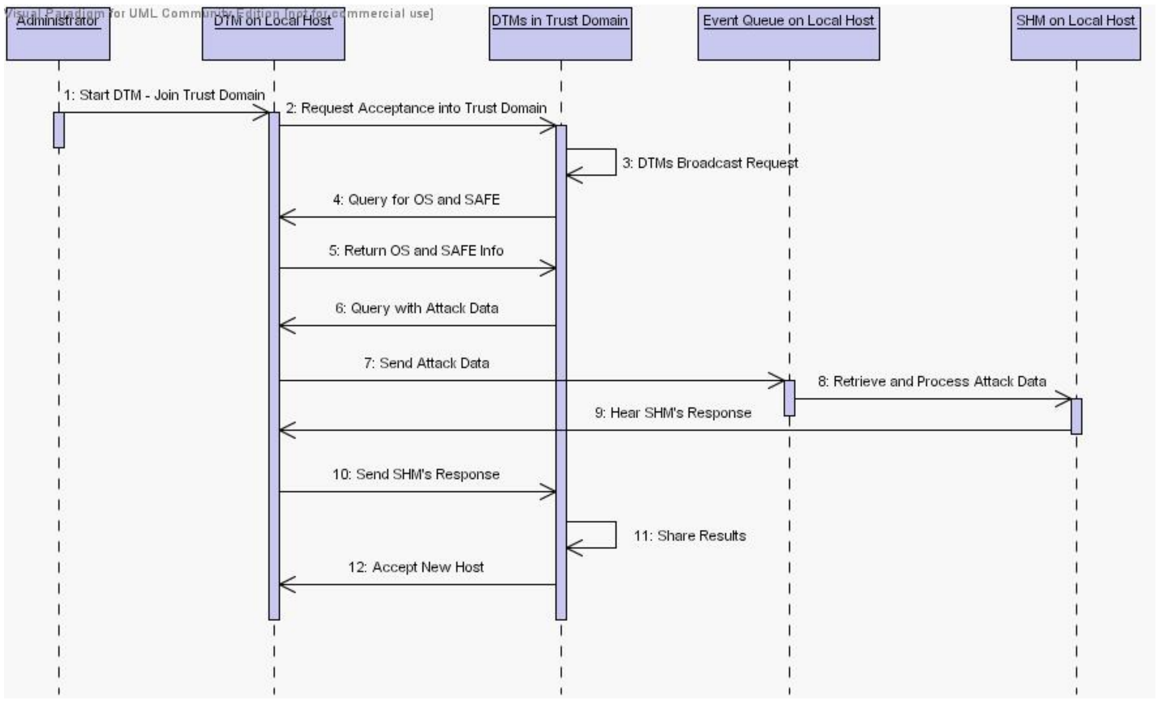


Sequence Diagrams

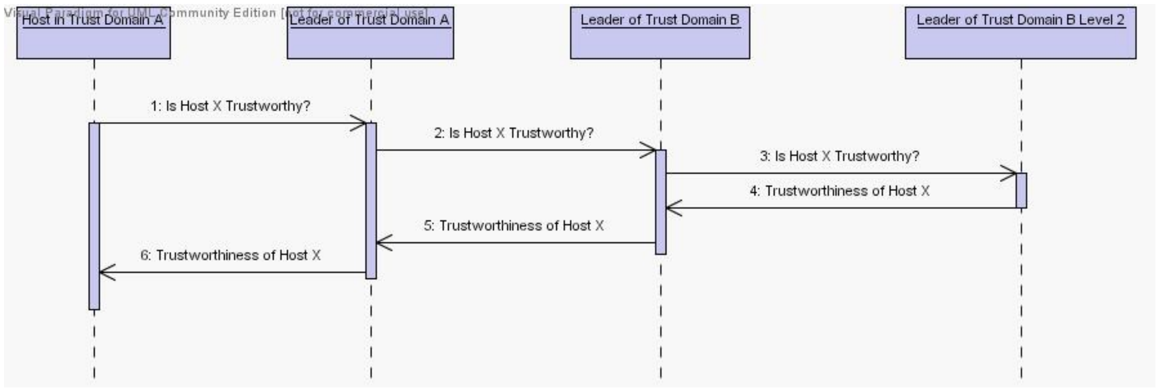
Starting DTM (and *S.A.F.E.*)



Joining a Trust Domain

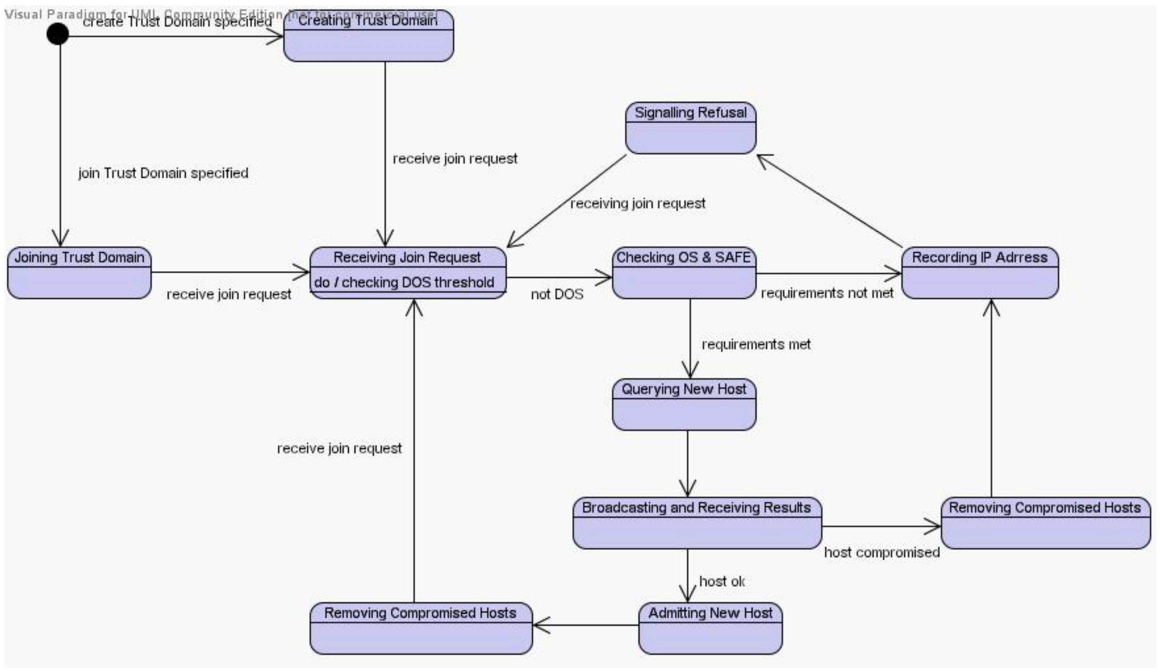

Visual Paradigm for UML Community Edition [not for commercial use]
Administrator
DTM on Local Host
DTMs in Trust Domain
Event Queue on Local Host
SHM on Local Host
1: Start DTM - Join Trust Domain
2: Request Acceptance into Trust Domain
3: DTMs Broadcast Request
4: Query for OS and SAFE
5: Return OS and SAFE Info
6: Query with Attack Data
7: Send Attack Data
8: Retrieve and Process Attack Data
9: Hear SHM's Response
10: Send SHM's Response
11: Share Results
12: Accept New Host

Communicating Between Trust Domains

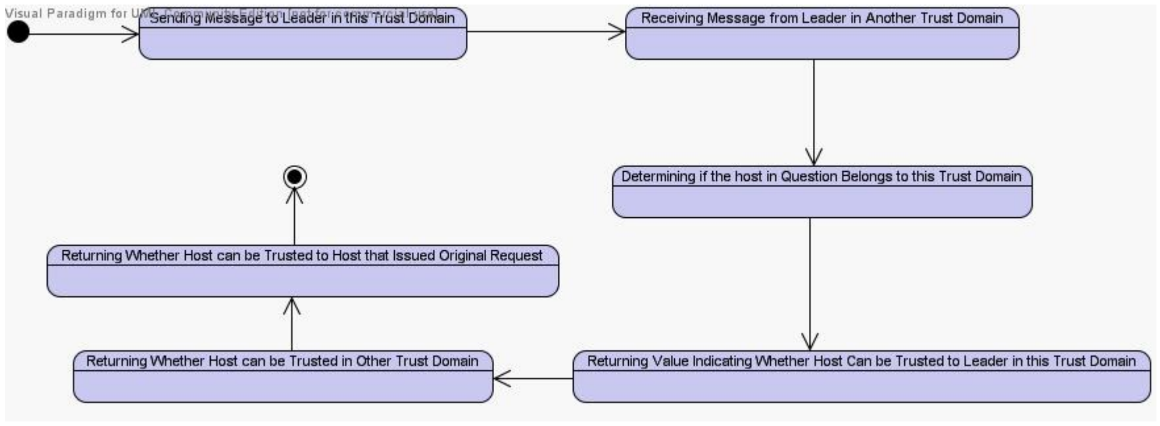

Visual Paradigm for UML Community Edition [not for commercial use]
Host in Trust Domain A
Leader of Trust Domain A
Leader of Trust Domain B
Leader of Trust Domain B Level 2
1: Is Host X Trustworthy?
2: Is Host X Trustworthy?
3: Is Host X Trustworthy?
4: Trustworthiness of Host X
5: Trustworthiness of Host X
6: Trustworthiness of Host X

State Diagrams

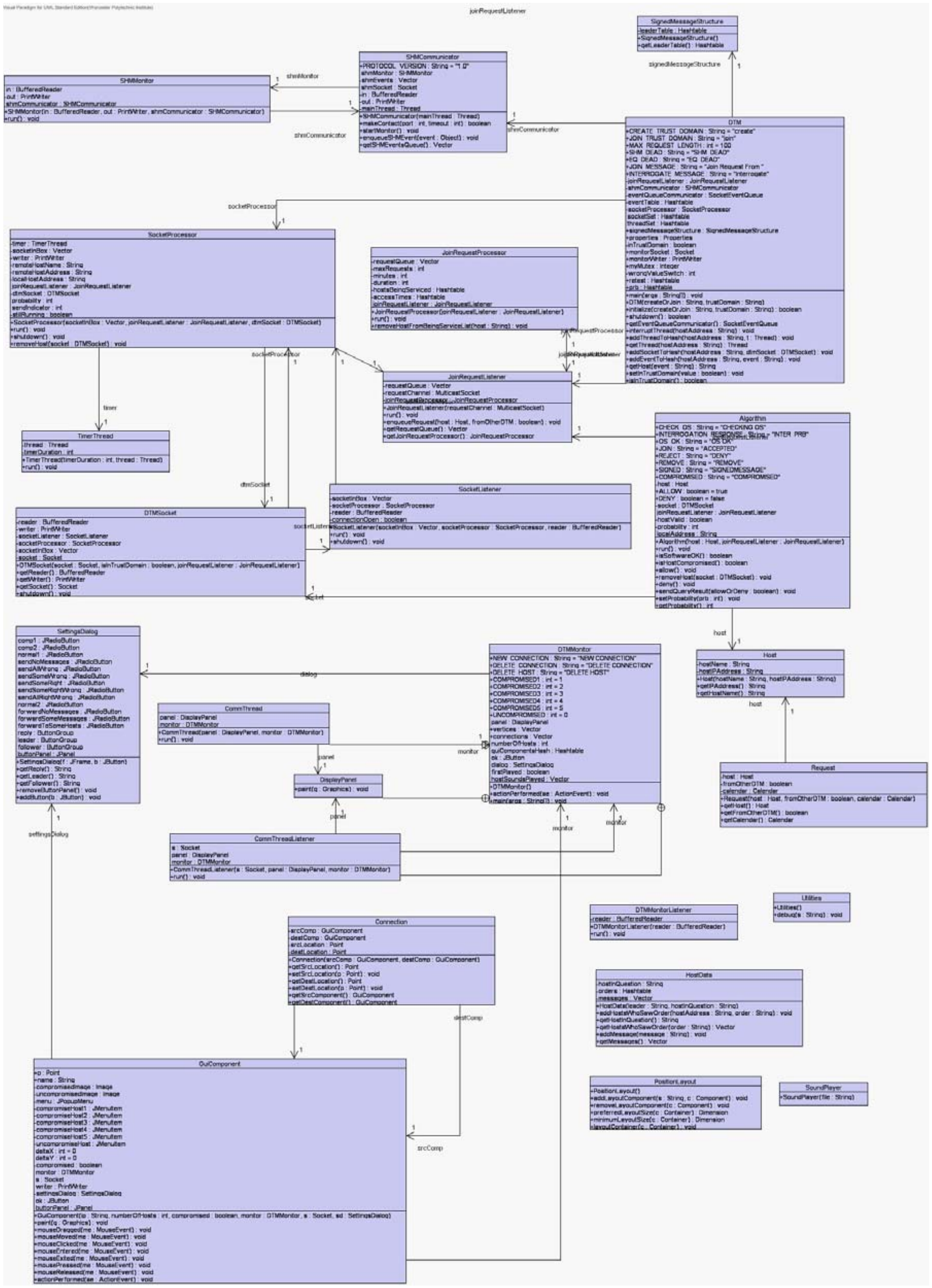
Joining and Maintenance of a Trust Domain



Querying whether a Node in Another Trust Domain is Trustworthy


Visual Paradigm for UML Community Edition [not for commercial use]
Sending Message to Leader in this Trust Domain
Receiving Message from Leader in Another Trust Domain
Determining if the host in Question Belongs to this Trust Domain
Returning Whether Host can be Trusted to Host that Issued Original Request
Returning Whether Host can be Trusted in Other Trust Domain
Returning Value Indicating Whether Host Can be Trusted to Leader in this Trust Domain

Class Diagram



3. Assumptions:

The following assumptions have been made. These should be addressed in a future phase of the project, if it continues.

We assume that the leaders of trust domains know which hosts are the other leaders of the other trust domains. We assume that the administrators of trust domains share this information off-line with a secure method. This assumption will need to be addressed in a future version of the DTM in order allow trust domains to communicate with each other with no human intervention.