MITIGATION OF NETWORK TAMPERING USING DYNAMIC DISPATCH OF MOBILE AGENTS

by

ADAM JAY ROCKE
B.S. University of Central Florida, 1998
M.S. University of Central Florida, 1999

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2004

Major Professor: Ronald F. DeMara

ABSTRACT

Detection of malicious activity by *insiders*, people with legitimate access to resources and services, is particularly difficult in a network environment. In this research, a novel classification of *tampering modes* by insiders against *Intrusion Detection Systems (IDSs)* is developed and addressed using distributed processing approaches. First, several *user capability ranks* and *tampering points* are identified to categorize critical exposures. Second, a tampering mode taxonomy including *spoofing*, *termination*, *sidetracking*, *alteration of internal data*, and *selective deception* is developed. Third, in response to these tampering modes, the *Collaborative Object Notification Framework for Insider Defense using Autonomous Network Transactions (CONFIDANT)* is developed and evaluated.

CONFIDANT employs interlocked mobile agents to reduce single point-of-failure exposures and increase barriers against insider tampering. While previous approaches relied upon monolithic architectures or agent frameworks using a centralized control mechanism or common reporting repository, they introduced distinct vulnerabilities. These vulnerabilities are identified in a novel hierarchy of IDS architectures. CONFIDANT realizes a *Distributed Control and Dynamic Dispatch (DCDD)* architecture using mobile agents for tampering detection, decision making, and alert signaling. It uses three *echelons* of agent interaction and four autonomous *behaviors* supporting *encapsulation*, *redundancy*, *scrambling*, and *mandatory obsolescence.*

The *Tampering Mode Exposure (TME)* metric weighting scheme is developed to compare CONFIDANT's response to that of the existing frameworks *Tripwire* and *AIDE.* Testing is performed to illustrate the mitigation techniques for each tampering mode using the Concordia mobile agent framework. Quantitative as well as qualitative metrics are assessed by dispatching Committees C^i of agents a^i_j where $1 \leq i \leq 2$, $1 \leq j \leq 12$ to perform filesystem scans and provide alarm notification. Test results indicate Tripwire's and AIDE's vulnerability to tampering via *Pacing*, *File Juggling*, and *Altering Internal Data* with TME scores of 65 and 59, respectively, out of a possible value of 123. CONFIDANT's DCDD framework achieves a score of 103 through mitigation of several exposures with the exception of *Processor Blockading.* These results demonstrate viable approaches for mitigating several challenging IDS exposures including many insider tampering risks.

ACKNOWLEDGMENTS

I thank the National Security Agency (NSA) for their financial support which helped make this work possible. Technical feedback from Dr. Richard Brackney at the NSA was especially helpful. As a recipient of the Lucent Florida Universities Fellowship, I also thank Lucent Technologies for the financial assistance with tuition, books, and travel expenses.

I also express thanks to my advisor, Dr. Ronald F. DeMara, for his many hours of guidance and support on this project. Additional thanks to Dr. Fernando Gonzalez, Dr. Guy Schiavone, Dr. Scott C. Hagen, and Dr. Aristide Dogariu for their time spent on my advisory committee. I also thank Dr. Simon Foo at Florida State University for his collaborative efforts and Mrs. Alexis Rossignolo for graciously proofreading this dissertation.

Special thanks and sincere gratitude to my parents for providing a nurturing environment in which I have been able to flourish.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Need for Secure Networking

In client/server environments, the number of points of entry into the network is growing rapidly with the increase in computers and other devices connected. Meanwhile, information systems in government and commercial sectors are highly interdependent due to connections via local area and wide area networks. While interconnection is valuable, increased connectivity introduces many potential avenues of attack [1].

Damaging intrusions can occur in a matter of seconds. In the 1980s, it was relatively straightforward to determine if an intruder had penetrated a computer system and to discover what operations were performed [2]. Now intruders hide their presence by disabling common services, by installing modified versions of system programs, and by modifying audit and log files using techniques that can be extremely difficult to trace [3].

Many attacks on computer networks do not hinge on exploits for flawed communication protocols or sophisticated cryptanalysis techniques [4]. The majority of security exposures in computer networks are a result of flaws in ordinary programs, such as text editors or Web servers. These common programs can be used by intruders to gain unauthorized access to resources in a computer network. Frequently, security exposures are due to improper configuration of such programs. Thus, recent developments in network security have focused on *identifying* intrusions in addition to *preventing* them.

Intrusion Detection Systems

The goal of an *Intrusion Detection System (IDS)* is to identify occurrences of security breaches capable of compromising the integrity of resources or services. An IDS can be an important component of defensive measures protecting computer systems from tampering. While intrusion detection should not be considered as a complete defense, it can play a significant role in overall network security [3].

Over the past 20 years, numerous research-oriented [5] [1] [6] [7] [8] and publicly-available [9] [10] [11] [12] [13] frameworks have been developed to mitigate a variety of intrusion exposures. Exposures range from the comparatively benign deployment of a Trojan horse that resets web browser home pages without permission, up through revenge by a system administrator who malevolently alters executables.

ID Goals

Intrusion Detection (ID) aims to positively identify all actual attacks without improperly identifying any non-attacks as malicious. Motivations vary for using ID technology to detect attacks. Some may be interested in collecting forensic information to locate and prosecute intruders. Others may use ID to trigger actions to protect computing resources. ID may also be used in order to identify and correct existing vulnerabilities [14].

A primary motivation is to provide a secure and reliable computing environment. A secure computer system is one that can be depended upon to behave as is expected while protecting resources from unauthorized access [15]. Networked systems have varying degrees of vulnerability to different forms of attack. Most computer networks employ some degree of access control as a first line of defense to protect resources [16]. These access controls strive to ensure that all access is properly authorized before performing the requested action. In that context, an IDS provides a tool to monitor both successful and attempted access within the network.

Types of Intrusions

Howard and Longstaff [17] have defined terms and a taxonomy to enable the exchange and comparison of computer security incident information. They define an attack as having five steps:

1. an attacker must use some tool (e.g. text or disk sector editor)
2. to exploit some vulnerability (e.g. program configuration)
3. to perform some action (e.g. authentication)
4. on some target (e.g. filesystem)
5. in order to achieve some unauthorized result (e.g. system resource availability for legitimate users or outsiders).

Intrusion detection systems do not provide preventative security measures, but rather are used as reactive mechanisms in conjunction with passive information assurance processes like firewalls and virtual private networks. In practice, an IDS attempts to detect attacks or attack preparations by monitoring either the traffic on a computer network, or application and operating system activities. Once intrusive behavior is detected, the IDS alerts a security administrator and may invoke an automated response such as closing down external communication paths or initiating a mechanism to trace the source of an attack.

Information provided by an intrusion detection system can help a security administrator to understand which systems were attacked and determine the steps performed during the attack. Damage control can be performed on the affected systems using this information. For instance, it may be possible to remove software planted by the attacker to facilitate later access to the system [18]. File integrity verification programs specifically enable a security administrator to determine potential malicious software installed by an attacker. Previous relevant IDSs with these types of capabilities are reviewed in detail in Chapter .

Focus on File Integrity Analysis

File Integrity Analyzers are a class of IDSs that automatically verify the content of security-critical files. Frequently referred to as *tripwires*, they attempt to detect if files have been modified in unauthorized ways. Once suspicious modifications are detected by triggering the tripwire, the analyzer may alert a security administrator or invoke some type of automated response. Alternatively, file analyzers can provide guidance for damage control, such as identifying the modified files needing to be restored or hooks installed by the attacker to facilitate subsequent access.

File integrity tools use one or more cryptographically-based hash mechanisms such as SHA-1 or MD5 [15] to compute a digest checksum for monitored files. Digests are subsequently recomputed and then compared against previous values to detect if the contents of the file have been modified. Ten of the most widely installed file integrity analyzers are reviewed in [19]. One of the most popular is commercially marketed under the name *Tripwire* [9]. It utilizes a *policy file* under the control of an administrator to describe the expected behavior of system and data files. The policy file identifies files that are expected to change and the types of changes permitted to each file in order to preclude the misidentification of anticipated changes as tampering. Upon initial execution, a *baseline database* is created according to the policy file using cryptographic hash functions. Subsequent scan operation is illustrated in Figure 1. During scan operation, the policy file contents are obtained to determine the monitored system files that are to be inspected. Hash functions compute a digest checksum for the first specified file. The result is compared to the value stored in the baseline database. This process is repeated for all policy file entries. Once all specified files have been processed, a report is generated containing any discrepancies encountered during operation.

In a networked environment, *Tripwire for Servers* executes on each node [20]. The *Tripwire Manager* [21] interacts with the Tripwire servers via Secure Socket Layer (SSL). This form of centralized policy management enables an administrator to define a single policy and distribute it to many similar systems across the enterprise. Additionally, file attributes such as owner and access modes are checked for inconsistencies. Tools such as *AIDE* [12], *Veracity* [13], and *integrit* [22] operate similarly.

File integrity checking is not a cure to a security problem, but it can help determine if files have been altered, removed, or added. In the event of an intrusion, this information can decrease the amount of time spent determining what was altered. File integrity checking alone will not secure a system but can improve overall security capabilities.

From a high-level perspective, an IDS monitors actions in the computing environment in order to identify possible signs of an attack. Since file integrity analyzers may perform periodic inspections, some file integrity frameworks may not be considered full-fledged IDSs [23]. Still, they occupy an important role as

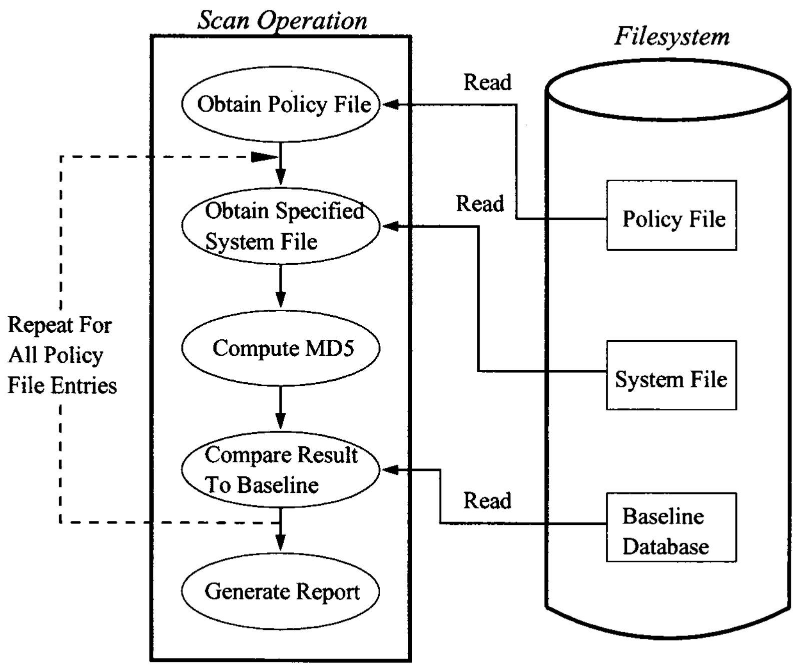

Figure 1: File Integrity Scan Operation

vital components of an intrusion detection environment. In this dissertation, the rationale for concentrating on the file integrity problem includes several compelling motives:

- cryptographically-hashed file integrity verification is the most popular tampering detection approach among installed systems,
- integrity verification approaches can positively detect many foreseen and even some unforeseen intrusions,
- regardless of the initial intrusion pathway, intruders often open a backdoor by altering system executables to facilitate future access [24],
- integrity analyzers can locate altered data so they become second-line targets of attack, and

- rudimentary alterations can defeat most existing file integrity analyzers.

While some file analyzers have taken steps to reduce tampering exposures, mitigation of risks from knowledgeable insiders remains as a challenging area. For instance, AIDE product literature warns that its own integrity cannot be guaranteed as AIDE's binary and/or baseline database can also be altered. In fact, their website warns that a hacked version is being maliciously distributed. One alternative recommended during Tripwire installation is to record the tool's binary files on write-once media. However, media within control of the system administrator become vulnerable to exchange. To address these concerns, tools such as Tripwire encrypt their baseline and verify its congruence with the policy file used to generate it. Tripwire also utilizes SSL communication protocols and triple DES encryption for critical transmissions. While these techniques can mitigate some risks, serious exposures from insider tampering have remained largely unaddressed.

Insider Tampering Exposures

A malicious action by a legitimate user, referred to as *insider tampering*, is particularly challenging to deal with. Insiders such as system administrators have broad access to sensitive resources, an extensive understanding of internal procedures, and frequent opportunities to carry out unauthorized use. Thus, attacks perpetrated by knowledgeable insiders have the potential to be more devastating than those that are externally-originated. Moreover, a common tactic of outsiders is to obtain limited access, then elevate their privilege to that of an administrator with a high capability levels. For this reason, insider risk is recognized as an exposure where few useful tools exist and significant exposures receive little attention [4] [25] [26].

Within the academic community, the insider problem is recognized as a difficult one. Neumann and Porras classify the detection of hitherto unknown attacks as very challenging open problems, citing subtle forms of misuse by insiders as a particular concern [8]. Recent DoD Workshops have identified the need for insider threat models as urgent [25]. In terms of a likely target of tampering within the domain of insiders, Axelsson identifies determination of the nature of attacks on the intrusion detection components as a fundamental unanswered question [27].

The conventional approaches to dealing with insider tampering are limited. They typically require concurrent login from two or more trusted individuals before granting administrator-level privileges. However, under operational conditions one individual may leave his/her station unattended at some point in time after simultaneous login. Even if this policy is abided by completely, it is unreasonable for two administrators with different backgrounds and familiarization levels to be fully cognizant of the rationale and implications of each action the other undertakes.

New Technologies to Apply

Existing solutions for network security management lack flexibility, adaptability, and autonomy. Therefore, it is useful to review the underlying techniques with which IDSs are designed and to consider new alternatives. In this context, *multi-agent systems* can provide a configurable balance among security requirements while maintaining platform flexibility and adaptability.

Agent Overview

The *actor* model of computation was originally proposed by Hewitt [28]. He defines actors as being self-contained, concurrently interacting entities of a computer system that communicate via message passing. Also, actors can be dynamically created and the topology of an actor system can change dynamically. Mobile agent technologies exemplify an actor paradigm for distributed computing for existing client/server platforms. In a mobile agent environment, programs travel between hosts in a network as required to achieve their execution requirements.

According to Agha [29], these approaches are formulated around three main design objectives:

- shared and mutable data,
- reconfigurability, and
- inherent concurrency.

Agents are self-contained components of a computing system that communicates by asynchronous message passing, as distributed agents have local clocks that may proceed at different rates. Message delivery is guaranteed to provide fairness, and an agent can send messages to activate other agents. These capabilities can be enhanced by relocating the required code to the platform where the data already resides.

Mobile Code Capabilities

Code mobility is the capability to dynamically change the bindings between code fragments and the location where they are executed [30]. Code mobility is not a new concept as research on distributed operating systems has focused on support for the migration of active processes and objects at the operating system level [31]. Process migration deals with moving an executing process from one node in a network to another. Process and object migration address the issues that arise when code and state are moved among the hosts of a loosely coupled, small-scale distributed system [30]. In this research, a mobile agent is a software component that is able to move between, and operate in, different execution environments.

Using Agents for ID

Mobile agents can address the capacity, scalability, and efficiency limitations of existing *monolithic* IDS architectures. Whenever a new form of unforeseen intrusion is identified, a monolithic IDS has to be substantially rebuilt to handle it, which is not a straightforward task. Mobile agent systems can be better suited to achieve the following desirable characteristics for an IDS [32]:

- run continually with minimal human supervision,
- strive for incurring minimal overhead on the system where it is running,
- be able to adapt to changes in system configuration and use,
- be able to scale to a large number of hosts, and
- provide graceful degradation.

An intrusion detection system using a mobile agent framework can exhibit these characteristics. A single agent cannot adequately perform intrusion detection since its vision is limited to a small portion of the network. Multiple agents

Table 1: Advantages of Using Agents in Intrusion Detection

Configurability	Can be added or removed from a system without altering other IDS or OS components
	Compatible with heterogeneous networks
Extendability	When a new attack is identified, new agents can be developed and added
	Easier coordination between interacting components
Efficiency	Communication cost reductions
	Hide network latency
	Reduce network load
	Execute asynchronously and autonomously
Robustness	Can continue to operate in the presence of physical or logical modifications to the network environment
	Support fault-tolerant behavior
	Flexible architecture for distributed computation
	Suitable for mitigating insider risks

cooperating with each other, however, can provide powerful IDS capabilities across heterogeneous resources.

Since agents are independently running entities, they can be added and removed from a system without altering other components and consequently remove the need to restart the IDS [33]. Also, whenever any sign of a new form of attack is identified, new specialized agents can be developed, added to the system, and configured to meet a specific security policy [34]. Other advantages of using mobile agents in an intrusion detection system [35] are listed in Table 1 where Configurability and Extendability can be traded off for Efficiency and Robustness.

CONFIDANT Introduction and Goals

Tampering modes are attack pathways capable of corrupting an ID framework. Tampering modes present in network-based file integrity analyzers are introduced along with a *Collaborative Object Notification Framework for Insider Defense using Autonomous Network Transactions (CONFIDANT).* In agreement with the meaning of the word *confidant* as "a most trusted servant," the CONFIDANT framework aims at trusted detection of unauthorized modifications to executable, data, and configuration files.

CONFIDANT exemplifies a third-generation agent-based security framework, based upon previous experience from the TACH [36] and FICA [37] systems implemented at the University of Central Florida. Design of CONFIDANT is based on two major goals:

Goal-1: *Reduce single point-of-failure exposures in existing IDS frameworks*, and

Goal-2: *Increase barriers against insider tampering.*

These goals will be evaluated by:

1. identifying single point-of-failure exposures in IDSs to address Goal-1,
2. developing a taxonomy of insider risks to address Goal-2,
3. designing metrics and experiments to quantify performance against both Goal-1 and Goal-2, and
4. comparing performance of the proposed and existing approaches using these metrics.

Dissertation Outline

This dissertation focuses on developing a framework using dynamic dispatch and distributed control of mobile agents for file integrity verification. Applications and program configuration are security exposures in computer networks and can lead to unauthorized access. For reasons discussed in Chapter 2, this dissertation focuses on insiders, or administrators, who have legitimate access to the system but may be abusing their privileges. Contributions made to the field of IDS design and insider robustness resulting from this dissertation include a classification of insider tampering modes, an IDS architectural taxonomy, a mobile agent approach to ID, and a comparative metric evaluation scheme.

Chapter 2 describes user capability classes and IDS architectural vulnerabilities. IDS tampering modes are defined and then formalized using an abstracted model of sensor, control, and alarm mechanisms. Classes of tampering modes identified include Spoofing, Termination, Sidetracking, Altering Internal Data, and Selective Deception.

Chapter 3 provides a review of selected intrusion detection systems and file integrity frameworks within the context of these tampering modes. Conventional file integrity analyzers such as Tripwire and AIDE are addressed. Existing agent-based frameworks including AAFID, TACH, and FICA are discussed. A proposed architectural taxonomy of IDSs is presented. IDS metrics, including qualitative and quantitative measures, are described.

The CONFIDANT agent framework is defined in Chapter 4, including its agent's gateways, behaviors, echelons, and interactions. Agent dispatch and communication functions are defined and then illustrated by example handshaking scenarios. The CONFIDANT approach to mitigating the defined tampering modes is also developed.

Testing methodology and results are provided in Chapter 5, including performance results in the absence of tampering, agent network traversal, and evaluation of the defined goals. Test cases for the defined tampering modes and a comparison of Tripwire, AIDE, and CONFIDANT responses are included. A weighting scheme developed to compare IDSs is presented. Chapter 6 provides a results summary and outlines future work.

TERMINOLOGY FOR INSIDER RISKS IN NETWORKED ENVIRONMENTS

It is useful to first identify IDS architectural vulnerabilities and correlate them to user capability in order to categorize tampering exposures. Once IDS exposures are categorized, a framework can be designed specifically to mitigate each category. To attain a robust file integrity framework, it is necessary and sufficient to consider those vulnerabilities within the domain of systems administrators or *super-users.*

Capability-driven Design Flow

Let TM_{SU} denote the set of *tampering modes* available to *super-users*, let TM_{LU} denote tampering modes of *legitimate users* without super-user capabilities, and let TM_O denote tampering modes of *outsiders.* By definition, $TM_{SU} \supseteq TM_{LU}$ since super-users can perform all operations available to any other legitimate user. Likewise, $TM_{LU} \supseteq TM_O$ since being a legitimate user does not preclude conducting tampering activities available to outsiders. By transitivity, $TM_{SU} \supseteq TM_O$, hence tampering modes of super-users subsume vulnerabilities of both legitimate users and unauthorized outsiders.

Motivated by this subsumption relationship, a generalizable design flow for an agent-based IDS framework can be derived. As depicted in Figure 2, *Rank I* exposures denote only those tampering modes available exclusively to super-users; *Rank II* exposures denote any non-Rank I modes available to legitimate users, but not to outsiders; and *Rank III* modes denote any exposure that is not a Rank I nor Rank II exposure. More formally, let the set of exclusive exposures of *Rank x* be

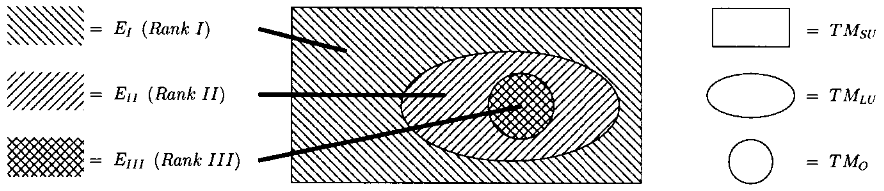

Figure 2: Relative Rank of Tampering Modes addressed by CONFIDANT.

denoted by E_x. Thus, $E_I = TM_{SU} - TM_{LU}$, $E_{II} = T_{LU} - TM_O$, and $E_{III} = TM_O$. Based on these rankings, the required agent behaviors can be developed as follows:

1. identify E_I vulnerabilities,
2. postulate an IDS design against which all identified vulnerabilities are evaluated,
3. repeat step 2 until a design is obtained capable of detecting all identified vulnerabilities,
4. identify E_{II} vulnerabilities,
5. verify the postulated design *already* meets all E_{II} vulnerabilities, or if it does not then return to step 2,
6. identify E_{III} vulnerabilities, and
7. verify the postulated design *already* meets all E_{III} vulnerabilities, or if it does not then return to step 2.

Using this design flow, many agent behaviors that were developed to detect only Rank I vulnerabilities were also capable of detecting Rank II and Rank III exposures. By focusing the agent development effort on super-user exposures, other vulnerabilities can be mitigated without the need to explicitly design mechanisms to address the lower rank exposures.

Clearly, super-users are the most capable adversaries. Thus, a proficient set of agent behaviors for mitigating E_I vulnerabilities can be powerful enough to address E_{II} and E_{III} vulnerabilities. By contrast, an agent behavior merely sufficient against an outsider is unlikely to be effective against a super-user. Previous IDS frameworks have focused on Rank III exposures and then later attempted to retrofit coverage for Rank II or perhaps Rank I exposures. However, as described in the following sections, CONFIDANT concentrates on addressing E_I vulnerabilities of insiders. By considering E_I vulnerabilities first, a capability hierarchy including E_{II} and E_{III} exposures is readily achieved.

IDS Architectural Vulnerabilities

The tampering capability classes describe categories of individuals based on access to computing resources. Physical tampering points in a computer system architecture along with the most general tampering capability at each point are listed in Table 2 and illustrated in Figure 3. Tampering by outsiders, denoted TM_O, without physical access to computing resources can only be performed remotely. Thus, TM_O is limited to tampering with network resources. In addition to the E_{III} exposure of tampering over the network, legitimate users are able to exploit E_{II} vulnerabilities including modify local filesystem contents and memory locations based on permissions assigned by the administrator. For this reason, tampering modes of legitimate users, denoted TM_{LU}, include points TP_{FS}, TP_{PT}, TP_{IC}, TP_{ID} in Figure 3. These stand for tampering points at the filesystem, process table, IDS code, and IDS data, respectively. Insiders have few restrictions on computer resource use. Administrators, unlike the other capability classes, can

Table 2: Computer System Resource Tampering Points

Tampering Point	**Definition**	**Associated Capability Class**	
		User Apps	**IDS Apps**
TP_{FS}	Alteration of filesystem contents	TM_{LU}	TM_{SU}
TP_{PT}	Modifying the process table in memory	TM_{LU}	TM_{SU}
TP_{IC}	Changing application code while in memory	TM_{LU}	TM_{SU}
TP_{ID}	Changing application data while in memory	TM_{LU}	TM_{SU}
TP_N	Tampering from remote network nodes	TM_O, TM_{LU}, TM_{SU}	
TP_{SC}	Modification of the system clock	TM_{SU}	

tamper with any filesystem resource or memory location as well as modify the system clock, shown as TP_{SC}.

In addition to physical tampering points, an IDS can be described in terms of *sensor*, *control*, and *alarm* logical components, as illustrated in Figure 4. Sensor components gather raw data from the system environment such as the contents of files being analyzed. Control mechanisms provide logic and decision-making routines to interpret the sensor outputs. Alarm subsystems respond to violations by implementing alert conditions. Tampering can occur at the host services and devices level, within each logical IDS component, and as input to each component.

IDS Exposures and Tampering Modes

Intrusion exposures have been classified in the literature based on numerous features [38]. These include the access pathway of the attacker, the system vulnerability exploited, and the procedures or data being targeted [17]. Classifications based on the intruder's intent have also been developed [39], as well as consideration of the intruder's knowledge level [40]. In this section, a widely-applicable classification is defined for exposures facing the IDS itself.

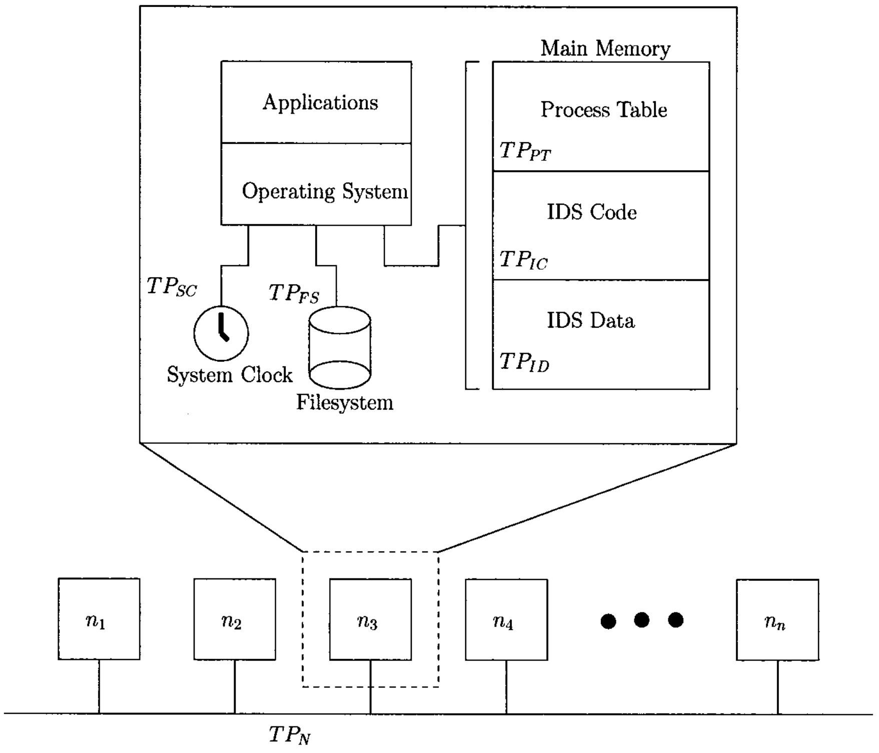

Figure 3: Physical Architecture

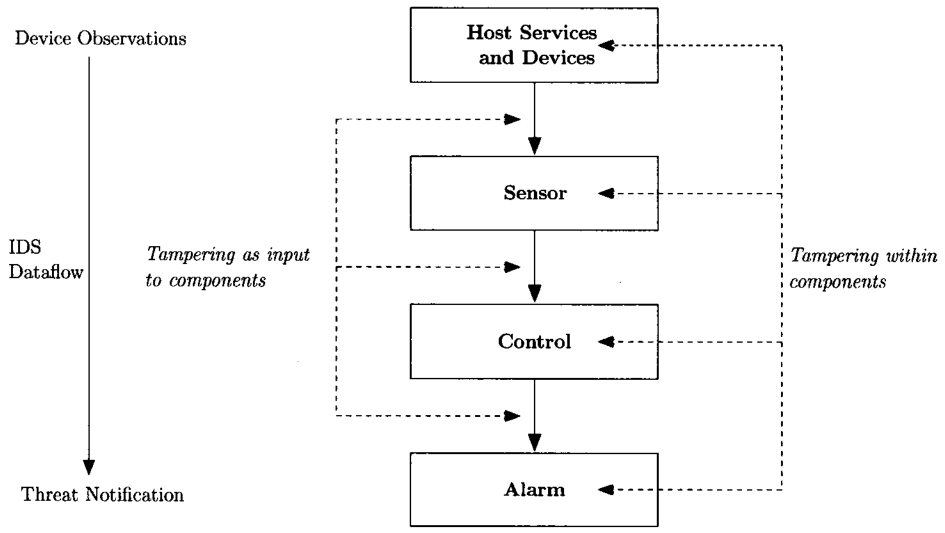
Device Observations
IDS Dataflow
Threat Notification
Tampering as input to components
Host Services and Devices
Sensor
Control
Alarm
Tampering within components

Figure 4: IDS Conceptual Architecture

Table 3: Tampering Modes Under Consideration

IDS Vulnerability		**Tampering Mode**	**Instantiation for File Integrity Analysis**
Spoofing data to:	Sensor	Spoonfeeding	Alternate data stream is conveyed during file scan
	Control	Sugarcoating	Unfavorable cryptographic digest is modified to appear as the desired result
	Alarm	Recanting	Fraudulent command is issued to deactivate alert
Termination of:	Sensor	Blindfolding	Detection mechanism is disabled
	Control	Commandeering	Decision-making process is usurped
	Alarm	Soundproofing	Notification mechanism is eliminated or muted
Sidetracking of:	Sensor	Blockading	Resource usage is forestalled to starve access
	Control	Pacing	Scan timing reference is corrupted or execution priority is overwhelmingly reduced
	Alarm	Scapegoating	Attention is diverted to a contrived distraction
Alter internal data in:	Sensor	Retroactive Baselining	Reference values for digests are modified
	Control	Descoping	Exemption is added to policy file to exclude scan coverage of unauthorized modifications
	Alarm	Value Jamming	Stand-alone process continuously writes `FALSE` into the memory location of status indicator
Selective deception		File Juggling	Target files interchanged before and after scanning

Tampering modes signify critical vulnerabilities for several reasons. First, their targets are the IDS facilities themselves that must be relied upon to detect adversarial events. Second, protection for the entire system may be compromised if tampering is successful. Third, crucial exposures to insider risk exist because an IDS framework is under the direct control of administrators.

An instantiation of each tampering mode for the case of file analyzers is listed in Table 3. Each logical IDS subsystem defined previously is vulnerable to *spoofing*, *termination*, *sidetracking*, and *internal alterations* as described by the tampering modes identified below.

Spoofing-based Tampering

Spoofing attacks transmit counterfeit data to mislead the recipient by tampering at TP_{FS} and TP_{IC}. An IDS sensor is vulnerable to spoofing by *Spoonfeeding* it information that is not present in the target file. Instead of accessing the file's real contents, an adversarial stream of data is provided in a compulsory manner. Data spoonfeeding occurs at and can be realized at the kernel level by hacking I/O routines to omit or insert information. Alternatively, a file request from the IDS may be redirected to an unmodified copy. When implemented successfully, these modifications are disguised by including their changes among those items being masked. For example, consider when IDS or kernel files are loaded for execution. Initially, suppose hacked versions get referenced. Later, the identical filenames may be presented for verification. However, these executables might be hacked to redirect any such read requests to the original versions instead. The adversary's capacity to distinguish `execute` system calls from `read` calls enables Spoonfeeding when necessary.

IDS control is vulnerable to *Sugarcoating* of unfavorable reports before evaluation. For instance, suppose that unauthorized modifications render an incongruent digest for a target file. However, the reports conveyed to decision-making routines are altered to provide digests for the unmodified file instead. Even if all intra-IDS communication is encrypted and secure, an exposure still exists. Modified IDS routines might sugarcoat reports about target files as well as incongruities about themselves.

The alarm subsystem is vulnerable to spoofing by *Recanting* alert notifications. For instance, a counterfeit notice to discontinue an alert might be sent ex-postfacto, as if the alert was only a false alarm. If the alert is recanted in a timely manner, then the alarm may not be properly realized or go unperceived.

Termination-based Tampering

Although more conspicuous, outright termination of IDS mechanisms can facilitate potential modes of tampering and are generally performed by process termination at TP_{PT}. As listed in Table 3, *Blindfolding* attempts to exploit this vulnerability. By terminating a sensor, the perceptive ability of the IDS may decline. Without proper safeguards, an IDS adversary might attempt to disable local sensor processes to obscure unauthorized file changes.

Termination exposures also include the *Commandeering* of IDS control. This refers to the overriding or bypassing of IDS decision-making operations. Commandeering of the IDS may involve removing critical routines from an executable. Hence, when unfavorable information is presented, the logic routines needed to detect intrusion are missing. For instance, suppose an IDS routine is stored on a host's hard drive in standard unencrypted executable format. It then becomes vulnerable to alteration using a disk sector editor. The critical opcode to bypass is the `Test_and_Branch` instruction selecting the 'intrusion detected' branch of the code. A discovery testbench that uses a divide-and-conquer approach to replace `Test_and_Branch` instructions and then observe the effect is not an implausible exposure. Sufficiently capable adversaries might also arrange for IDS modifications during loading or after the code resides in memory.

Termination-oriented vulnerabilities at the alarm level include *Soundproofing* techniques. Soundproofing an IDS framework involves muting the alarm to preclude end-user notification. This includes terminating the alarm process or halting inclusion of culpable entries in a report or database. Likewise, an IDS adversary may also disable alarm capability via bypassing of execution. For example, suppose the first instruction in an alarm routine is replaced with a `Return_from_Subroutine` opcode. The appropriate location may be listed in a symbol table within the executable. It might also be reverse engineered from among the `Jump_Subroutine` destination addresses and known system calls.

Sidetracking-based Tampering

More sophisticated, yet less detectable, tampering modes attempt to sidetrack the IDS. They interfere with file integrity operations through collateral means, such as Denial of Service (DoS) attacks against IDS mechanisms. *Blockading* attempts to isolate a sensor from needed access to a target file or device. Some integrity frameworks can be blockaded by not relinquishing exclusive non-preemptive privileges. Robust file verification involves taking into account that Blockading attacks are plausible against a wide range of IDS operations.

IDS control mechanisms are also vulnerable to sidetracking by *Pacing* techniques. Pacing alters the execution rate in order to decrease the effectiveness of the IDS. A direct way to pace a file analyzer is by increasing the period between integrity scans. Similarly, an adversary may attempt to postpone scanning by resetting the time-of-day clock by tampering at TP_{SC}. Implicit means such as demoting IDS execution priority can also be problematic. Either approach might be sufficient to provide an intruder with an undetected window of access.

At the alarm level, *Scapegoating* is an IDS tampering mode that focuses blame on unrelated events. For instance, an alarm may be intentionally triggered for a decoy cause in order to camouflage the actual attack. Similarly, a multiple alert

DoS attack may be launched that overwhelms the alarm's processing mechanism or intended human recipient. Once either of these becomes overloaded, the system is potentially exposed to undetected intrusions.

In general, sidetracking-based tampering modes are difficult to mitigate. They exploit vulnerabilities whereby file analysis is impaired, yet each IDS component remains installed, unmodified, and in operation. This implies mitigation of IDS sidetracking will require ancillary techniques beyond the file verification methods themselves.

Internal Data Tampering

IDS data structures residing on disk, TP_{FS}, or in memory, TP_{ID}, can also be vulnerable to tampering. *Retroactive Baselining* is the after-the-fact modification of reference values. For example, integrity checkers create baseline files that store the digests for the initial state of files. Retroactive Baselining corrupts these reference values. The updated baseline may reflect the digests after file modifications were made. Corresponding restoration of the baseline's access descriptors may help obscure the changes.

At the control level, an IDS is vulnerable to reductions in its operating scope to exclude analysis of malicious events. With respect to file analyzers, IDS *Descoping* amounts to excluding integrity verification of unauthorized changes. The range of verification may be descoped by tampering with the analyzer's policy file. Policy files are employed by integrity checkers to limit false positives. False positives correspond to file modifications that are routine, expected *a priori*, and not indicative of any malicious intent. So this tampering mode can involve appending entries to the exception list in the policy file. Normally, policy files are maintained by the system administrator which introduces a straightforward insider pathway. Similarly, there is exposure to the analyzer being spoonfed the adversary's version of the policy file.

With respect to alarm mechanisms, an IDS may be vulnerable to *Value Jamming.* This tampering mode employs an independent adversarial process. This high-priority process continually writes `FALSE` to a status flag maintained in memory. Through repeated jamming of a status value, it may be feasible to preclude sustained establishment of an alert. This underscores need for proper access control to memory pageframes for robust IDS operation.

Selective Deception

Selective deception refers to tampering modes that are colloquially known as double-dealing. By way of analogy, consider an unscrupulous casino dealer who selectively issues playing cards from two decks to defraud the recipient. One problematic form that impacts file analyzers is *File Juggling.* Realistically, file integrity verifiers can inspect target files only intermittently. So these tools are susceptible to tampering at TP_{FS} due to the existence of modified data at times other than file verification.

Restoration of authentic data immediately prior to scanning might be achieved through a *File Juggling script.* If an adversary is able to detect when a file integrity check will occur, then a modified file can be temporarily reverted to its original state. Suppose verification is set to occur with a pre-defined interval τ. The IDS may be susceptible to the File Juggling script shown in Figure 5. Line 1 provides access to the modified file 95% of the time, Line 2 restores the unmodified version prior to checking, Line 3 waits for the integrity scan to complete, and Line 4

```
    while(1) {
1:          sleep($95 \times \frac{T}{100}$);
2:          rename(unmodified_version,original_filename);
3:          sleep($5 \times \frac{T}{100}$);
4:          rename(modified_version, original_filename);
            }
```

Figure 5: Fixed-Interval File Juggling Script.

restores the modified version. More extensive operation would be necessary to restore the original file creation time and owner.

Alternatively, suppose that scans are scheduled at random intervals. Exposure to active detection of scanning operations would become a concern. In particular, the IDS adversary may replace Line 2 in Figure 5 with `(while not(exec("grep scan_process < top")))`. This creates a busy waiting loop until the `scan_process` begins executing, at which time files are interchanged. Fine-tuning of the File Juggling reaction time can be obtained by tweaking process priorities and resource blockades. In other words, a combination exposure exists from File Juggling, Pacing, and Blockading tampering modes. A discussion of exposures in the presence of combining individual tampering modes is provided below.

Summary

Existing IDSs exhibit a variety of exposures based on mechanism, user's knowledge, and access permissions. As shown in Figure 6, outsiders exhibit exposures of the lowest rank, Rank III, and are able to perform only Blockading across the network. In addition to Rank III exposures, Legitimate Users can also perform File Juggling in certain circumstances. Consider the case of delegating web server administration responsibilities to someone who is not the super-user. The webmaster has the required access to modify web server configuration. If these configuration files are

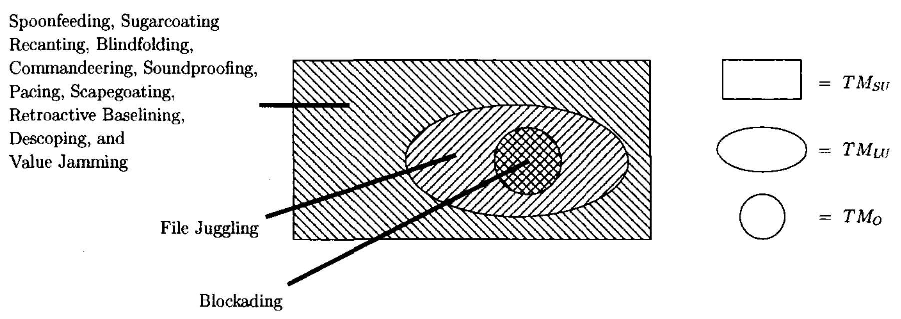

Figure 6: IDS Tampering Mode Rank

monitored as defined by the administrator in the IDS policy data, the webmaster has the ability to tamper via File Juggling. The remaining eleven exposures are restricted to Rank I, or the super-user level, thus motivating this research.

EXISTING FILE INTEGRITY FRAMEWORKS AND EVALUATION METRICS

Previous IDS and file integrity frameworks have relied on a client-server architecture to perform tasks ranging from file integrity analysis to user profiling. More recent agent-based approaches offer the potential of increased adaptability, reduced communication costs, and fault tolerance. The following discussion is restricted to file analyzers and multi-agent IDS frameworks.

File integrity tools use cryptographically-based hash mechanisms to compute a signature for monitored files. Signatures are subsequently recomputed and compared to previous values in order to detect if a file has been modified. A number of the most widely installed file integrity analyzers are reviewed in [19] and [41]. The relevant features of a few of them are discussed below.

Conventional File Analyzers

Selected file analysis tools are listed in Table 4. While *Tripwire* [9] is the most popular commercial file integrity analyzer, other commercial and open source alternatives operate similarly. Tripwire utilizes a *policy file* to describe the expected behavior of system and data files, identify files that are expected to change, and the types of changes permitted to each file. A *baseline database* is created using hash functions according to the policy file as a reference to detect file modifications. In a networked environment, Tripwire installed on individual hosts can interact with a *Tripwire Manager* via Secure Socket Layer (SSL) which provides message encryption. Tripwire Manager's form of centralized policy management enables an administrator to define a single policy and distribute it to many similar systems

Table 4: Selected Conventional File Integrity Analyzers

IDS Framework Name	**Availability**	**Execution Model**	**Comment**
Tripwire	Commercial and Open Source	Client-Server	The most popular commercial file integrity analyzer
AIDE	Open Source	Single Host	Created as a response to the commercialization of Tripwire
integrit	Open Source	Single Host	Designed to be a simple alternative to Tripwire and AIDE
Veracity	Commercial	Client-Server	Concentrates on manipulating snapshots of directory trees
Nabou	Open Source	Single Host	Can be used as a process monitor
SMART Watch	Commercial	Single Host	Detects changes without relying on periodic timers

across the enterprise. Other file analyzers such as *AIDE* [12], *Veracity* [13], and *integrit* [22] operate similarly, although AIDE aims toward removing particular limitations in Tripwire, while integrit focuses on essentials. Other existing tools exhibit unique features. For instance, Nabou [42] can be used as a process monitor, while SMART Watch [43] detects file system changes in near-real time by not using periodic timers.

While some file analyzers have taken steps to reduce tampering exposures, mitigation of risks from knowledgeable insiders remains as an evolving area. For instance, AIDE product literature warns that its own integrity cannot be guaranteed as its binary and/or baseline database can also be altered. In fact, their website [12] warns that a hacked version is being maliciously distributed.

As stated previously, some file analyzers have taken steps to reduce tampering exposures, but mitigation of insider risk remains a concern. One technique to mitigate tampering as recommended by Tripwire is to record binary files on read-only media. However, even write-once media within control of the system administrator become vulnerable to exchange. To further address these concerns, tools such as Tripwire encrypt stored binary data. Tripwire also utilizes SSL communication protocols and triple DES encryption for critical transmissions. Yet, these tools are still highly subject to several straightforward tampering modes. A complete discussion of the modes by which tampering can occur are presented in the previous section.

Agent-based IDS and File Integrity Approaches

Recently agents have been proposed as a technology to overcome limitations in a variety of intrusion detection applications beyond file integrity. Rationale for considering agents in an IDS ranges from increased adaptability for new threats to reduced communication costs. Since agents are independently executing entities, there is the potential that new detection capabilities can be added without completely halting, rebuilding, and restarting the IDS. Other potential advantages are described by Jansen [35], and by Kruegel [44] who also identifies downside tradeoffs including increased design and operational complexity. Use of mobile code itself introduces new security exposures and need for mitigation methods as discussed in [45]. Although numerous agent-based IDS frameworks have been proposed or constructed, few have aimed at addressing exposures from insider tampering as described below.

Agent-based Anomaly Detection

The *Autonomous Agents for Intrusion Detection (AAFID)* framework [11] developed at Purdue University is an IDS project employing autonomous agents for data collection and analysis. AAFID utilizes *agents* hosted on network nodes, *filters* to extract pertinent data, *transceivers* to oversee agent operation, and *monitors* to receive reports from transceivers. These entities are organized into a hierarchical architecture with centralized control. While a transceiver may report to multiple monitors to provide redundancy, monitors as well as AAFID agents remain at a single physical address upon deployment.

Cooperating Security Managers (CSMs) [5] enable individual distributed intrusion detection packages to cooperate in performing network intrusion detection without relying on centralized control. Each individual CSM detects malicious activity on the local host. When suspicious activity is detected, each CSM will report any noteworthy activity to the CSM on the host from which the connection originated. The local CSM will not notify all networked systems, but rather only the system immediately before it in the connection chain.

Other agent-based hierarchical architectures include the Intelligent Agents for Intrusion Detection project at Iowa State University [46] with a centralized data warehouse at the root, data cleaners at the leaves, and classifier agents in between.

Bernardes and Moreira have proposed a hybrid framework with partially distributed decision making under the control of a centralized agent manager [34]. Agents are deployed to observe behavior of the system and users. Agents communicate via messages to advise peers when an action is considered suspect. The architecture is structured into four distinct layers. When an agent considers an activity to be suspect, an agent with a higher level of specialization for the suspected intrusion is activated. Agents then report their findings to a centralized manager.

In these systems, distribution of some aspects of the data collection and decision-making processes suggests important features to help diffuse some tampering points. Yet, whenever agents are managed from a common server, identifiable targets for tampering remain. Furthermore, the use of one or more centralized repositories leave at least some portion of the network exposed to denial of service attacks.

Even if an autonomous mobile decision-making agent was to detect a problem, interlocking mechanisms would be necessary to preclude its accidental or malicious removal, delay, or spoofing.

Mobile Agents Supporting Remote-Access Detection of Misuse and Viruses

The University of Idaho has developed the *Hummingbird* framework [47] for managing misuse data. Hummingbird agents are neither autonomous nor mobile but do illustrate important methods to mitigate tampering such as including validated transactions between stationary decision-making centers, redundant data collectors, and use of Kerberos. The project and test cases used focus on cooperative intrusion detection if sharing of data is a viable option between distinct hosts across an enterprise.

The *Tethered Agent and Collective Hive (TACH)* concept for remote-access system management was defined by Costa [36] at Lockheed Martin in 1998. Figure 7 shows a high-level view of TACH. The three components of the architecture include a centralized *Hive* to keep track of agents and collected data, a *Task Manager (TM)* to assign priority codes and conditions of task execution, and an *Agent Registry (AR)* to track fingerprints of agents. Working with Lockheed, the University of Central Florida (UCF) designed and developed an Aglet-based [48] framework for TACH including mobile agents for virus detection [49] [50] and misuse detection [51]. Upon deployment by the TM, agents establish communication with the Hive, and the AR registers the agent as "live", then executes tasks as defined by its *Customized Task Module (CTM)*. A CTM is a plug-in that allows a standardized TACH framework to be more readily customized to new agent behaviors without

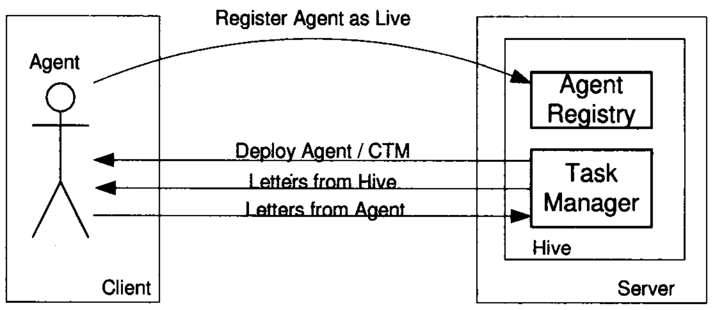

Figure 7: TACH Architecture

the need to modify other components or significantly change the communication protocols between agents and the Hive. Limitations of TACH include the use of a centralized Hive for agent control and a periodic communication protocol between the Hive and agents with time-out detection used to assess status changes. If the Hive is disabled then the entire TACH system would be compromised.

File Verification

Based on experience with TACH at UCF, the *File Integrity using Cooperating Agents (FICA)* framework [37] was developed using Concordia mobile agents, MD5 protocols, and Java APIs. FICA deploys two agent behaviors: an *initiator* to travel to remote nodes to create a baseline and an *examiner* to compute new digests against the original baseline. As with other file integrity analyzers, FICA must rely upon write-once media for configuration files and immutable digests. Also, since a centralized server for dispatching is employed, serious exposures remain.

IDS Architecture Taxonomy

Based on the previously described exposures, certain IDS design characteristics are utilized by CONFIDANT to meet Goal-1 and Goal-2. For instance, tampering complexity increases if functionality is not bound to a physical location. Also, unpredictable scheduling of IDS events increases the difficulty by which certain events can be anticipated and thus prevented, or spoofed by an insider. One technique to assist in avoiding a single point-of-failure exposures is to distribute IDS control across network nodes so that compromising a single node will not fatally disrupt execution on other nodes. Based on these characteristics, the following taxonomy of IDS architectures is proposed.

Centralized Control, Static Dispatch Architectures

Centralized Control with Static Dispatch (CCSD) architectures are the most fundamental configuration of IDSs. Architectures of this type restrict data and control functionality to a single machine and have a static network topology as illustrated in Figure 8. An example of this category is any single-host IDS, such as Tripwire or AIDE execution on a single machine with a local baseline database. Here the single machine controls the IDS and all data remains local to each machine. If an intruder is able to compromise the single machine, the IDS can be compromised. An example of a multi-host CCSD IDS is the Tripwire Manager controlling multiple instances of Tripwire for Servers.

Distributed Control, Static Dispatch Architectures

In order to overcome limitations in CCSD architectures and provide functionality for multiple network nodes, control functionality and data can be distributed across

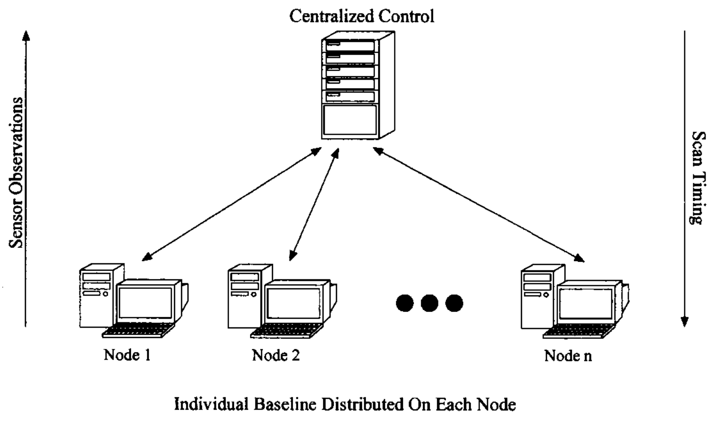

Figure 8: Centralized Control, Static Dispatch Architecture

a network domain. *Distributed Control with Static Dispatch (DCSD)* architectures are more sophisticated than CCSD due to the inclusion of multiple machines with control responsibilities in a distributed network, as shown in Figure 9. IDS control is distributed as individual machines are responsible for local computations. The dispatch of information is static as a central management database node records data served by individual nodes. Data dispatch is static due to a well-defined network topology and communication between client nodes and the central database. If an attacker is able to control the centralized management database, the entire IDS can be compromised. It is possible that the distributed management consoles are unable to receive accurate signature data and alerts generated by the client nodes, so DCSD architectures of this type exhibit a single point-of-failure. Hummingbird is an example of a DCSD architecture as it maintains a database for storing misuse data. While CSM does not maintain a misuse database, interaction between CSMs on each host is static.

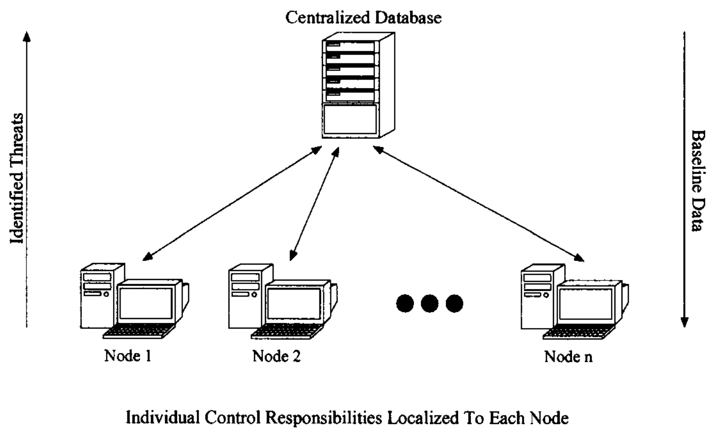

Figure 9: Distributed Control, Static Dispatch Architecture

Centralized Control, Dynamic Dispatch Architectures

Centralized Control with Dynamic Dispatch (CCDD) architectures have a single control module coupled with dynamic data movement. An example of this type of architecture is FICA [37] illustrated in Figure 10. In the FICA framework, agents are dispatched across the network to collect data and return only the filtered results to a single host for processing. This single host serves as the centralized control node. Since the agents can be sent to any node in the network at times specified by the server during runtime, there is a dynamic dispatch of agents. There is also a dynamic dispatch of data as data travels with the agents throughout the network domain. As with the previous two types of architectures, a single point-of-failure vulnerability exists. When client nodes rely on a single control server, modifications to control instructions can alter scan timing and produce erroneous scan results. In the case of mobile agents, whenever agents report to a single physical address that becomes compromised, resources are no longer protected. The hierarchical

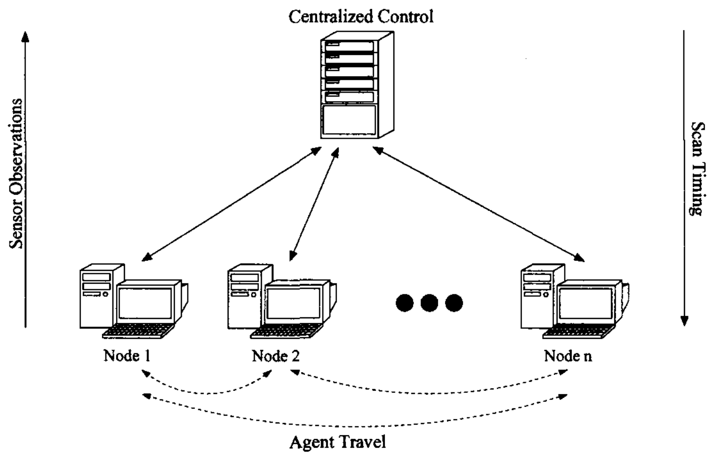

Figure 10: Centralized Control, Dynamic Dispatch Architecture

structure of distributed IDS elements such as agents, transceivers, and monitors is subject to compromise.

Distributed Control, Dynamic Dispatch Architectures

The final architecture defined is *Distributed Control with Dynamic Dispatch (DCDD)*. CONFIDANT exhibits an architecture of this type. This novel architecture class diffuses single point-of-failure exposures common to the other architectures by distributing control of the IDS, as well as providing a dynamic dispatch of data contained in agents. Control and data are distributed across a completely connected network domain. By distributing both data and control functionality, the single point-of-failure in other architectures is eliminated.

A hierarchy of the architecture classes defined in this section is given in Figure 11. As shown in the figure, class A is said to subsume class B if the mechanisms in class A can perform the operations of those in class B. For instance, when FICA (CCDD) either doesn't dispatch agents or dispatches agents to the local

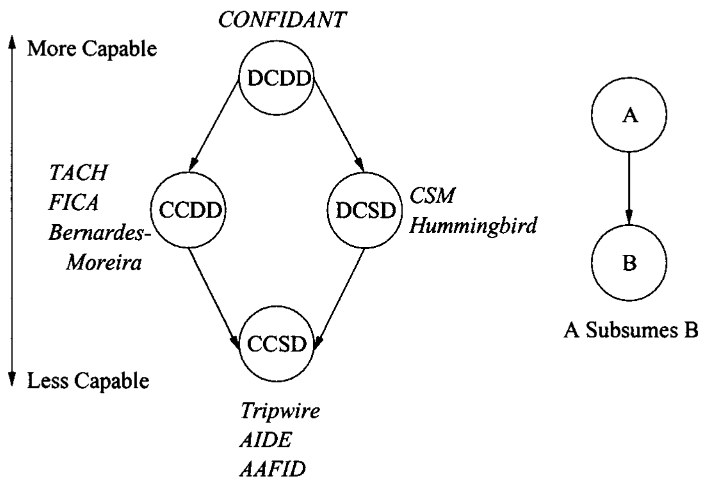

Figure 11: Hierarchy of Architecture Classes

machine, it operates as a CCSD architecture. A DCDD architecture such as CONFIDANT can function as a CCDD architecture if agents are required to report to a single centralized host. If a DCDD architecture can be realized, then it will provide a general technique for intrusion detection capable of mitigating additional vulnerabilities. Chapter will describe how a DCDD architecture is realized in CONFIDANT. To assess the performance of CONFIDANT's DCDD realization, a number of comparative metrics are developed in the following section.

Existing IDS Metrics

Evaluation of Intrusion Detection Systems can be challenging as many objective and subjective issues must be considered. An identified intrusion may be an artifact of an older attack or be properly detected, but provide insufficient diagnostic information to address the source of intrusion. Accuracy and avoidance of false alarms are important considerations due to their increase in severity and administrator workload.

Constructing broad, accurate benchmarks is particularly difficult due to the complex operating environment of an IDS. Ranum [52] describes the subtle difference between quantitatively measuring systems against a defined and predictable baseline and comparing systems directly to each other. This distinction is important as measuring against a baseline is more difficult to implement while comparative measurements lack repeatability. As a result, IDS testing tends to be either comparative or subjective, focusing on the accuracy or the quality of results.

Injection of Synthetic Network Traffic

Synthetic network traffic is often introduced to facilitate IDS evaluation. However, injection of synthetic traffic can be counterproductive as flooding the network with traffic is not sufficient for IDS performance evaluation. Production networks do not carry random traffic unless they are under attack or severely broken [52]. For instance, hosts are not subject to invalid frames that do not belong to established TCP sessions unless they are being brought under a denial of service attack. An evaluation involving a denial of service attack may cause a less rigorous IDS appear to perform better than one that extensively analyzes packets.

IDSs also respond differently to various types of network traffic, as some network-based IDSs evaluate only the header portion of the packet, while others will analyze the data portion as well. Synthetic traffic is sufficient for testing network hardware throughput, but not for IDS robustness evaluation [53]. Furthermore, hardware such as routers and firewalls exhibit a filtering effect on network traffic [52] and thus reduce the amount of synthetic traffic observed on associated network segments.

One worthwhile approach is to perform simultaneous tests involving production networks and real traffic from the site where the IDS is to be deployed. This involves configuring each IDS, then performing simultaneous evaluations. Another approach is to generate synthetic loads with standardized payloads. The most realistic approach for comparative testing involves a production network running attacks to check IDS responses [52]. Evaluations must be performed multiple times, as IDS operation in a realistic environment may produce varied results due to factors such as the states of the attacking machine [54].

Metric Classes

IDS evaluation involves a combination of both quantitative measurements and qualitative attributes. The combination can be either a defined algorithm or expert analysis with subjective judgment. Evaluation functions can be more straightforward to specify than the corresponding process. The process description is important to promote repeatability [55]. Requirements for IDS testing can even differ between architectures. For instance, in testing real-time systems, emphasis should be placed on speed and accuracy. For distributed systems, emphasis should be placed on minimizing false negatives while accepting increased false positives.

Quantitative Metrics

The ultimate measure of an IDS is the ability to detect intrusions within a reasonable time period. Secondary measures include the number of false positives and the ability to correlate data from multiple sources. An approach to accurately evaluate complete systems involves a predictable number of attacks coupled with measuring results returned by the IDS [52].

A valid detection is referred to as a True Positive Intrusion (TPI) and occurs when an alarm is sounded in the presence of an intrusion. A False Positive Intrusion

Table 5: Intrusion and Alarm Nomenclature

	Presence of Intrusion: I	**Absence of Intrusion:** $\neg I$
Presence of Alarm: A	True Positive Intrusion	False Positive Intrusion
Absence of Alarm: $\neg A$	False Negative Intrusion	True Negative Intrusion

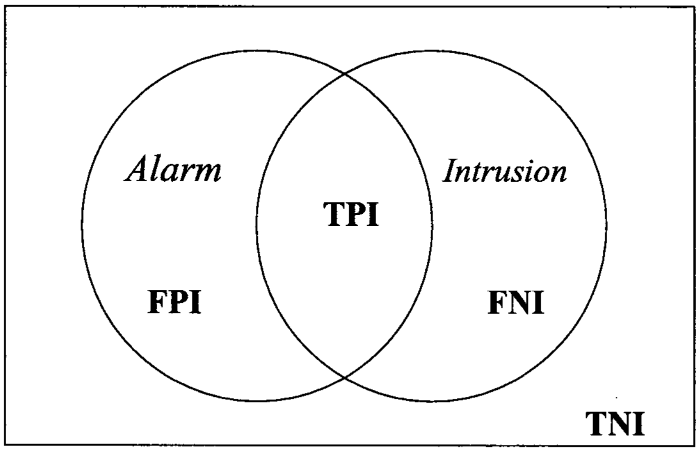

Figure 12: Quantitative Metric Relationship

(FPI) occurs when an alarm is triggered, but no intrusion is present. The number of false positives can be computed by subtracting the number of correctly detected attacks from the number of alarms raised. The absence of an intrusion that is correctly identified is referred to as a True Negative Intrusion (TNI) while a False Positive Intrusion (FPI) occurs when an alarm is triggered in the absence of an intrusion. The relationship between these four numerical measures is in terms of the presence or absence of an intrusion, denoted as I and $\neg I$, vs the presence or absence of an alarm, denoted as A and $\neg A$, is listed in Table 5 and illustrated in Figure 12.

Qualitative Metrics

An important consideration when using qualitative metrics for evaluation is the IDS class. A particularly complete set of metrics is provided in [53]. The defined metrics are divided into three categories: *logistical*, *architectural*, and *performance*. Logistical metrics measure IDS manageability including ease of configuration and management. Architectural metrics compare how the IDS matches the deployment architecture. Selected architectural metrics include adjustable thresholds and support for multiple sensors. Performance metrics include interaction with firewalls and routers and timeliness.

Computation of the weighted overall score for metric class S_j is listed in equation 1 where j corresponds to the three metric categories defined previously, i is the metric index within the jth category, n_j is the number of metrics within category j, U_{ij} is the unweighted score for metric i of category j, and W_{ij} is the weight of the $ijth$ metric.

$$S_j = \sum_{j=1,3} \left[\sum_{i=1,n_j} (U_{ij} * W_{ij}) \right] \tag{1}$$

The two methods for measuring each metric are observation analysis and review of material such as specifications or white papers. Individual metrics are assigned a low, average, or high score and weights are selected according to the intended IDS environment. System requirements must be clearly defined in order to define appropriate weights. The authors specifically warn that assigned weight values will always be subjective, but the method can be applied consistently as long as goals are accurately and uniformly defined.

Role of Numerical Measures

Certain defined formal metrics are difficult to observe. For instance, an administrator may not be aware of an IDS failing to detect an attack, so the number of false negative intrusions may be inaccurate. This can be overcome by using known traffic with standardized content. It is also possible that a particular classification may detect a series of events as a single attack, while another may see several attacks [53].

An excessive number of false positives can cause administrators to ignore IDS alerts. In order to compensate for a high number of false positives, an IDS that ignores actual intrusions will provide a false sense of security [53]. Consider the case where an IDS signals an alarm on every monitored operation. The detection rate will be 100%, but the number of false positives will be maximal as well [52].

Significance of False Alarms

It is suggested that there is no such thing as a false alarm in terms of computer security as any alarm contains vulnerability information [56]. While some Computer Incident Response Teams request that all data be disclosed even in the presence of false alarms [57], the number of false alarms can become overwhelming.

Axelsson [58] describes this as the base-rate fallacy problem. The implication of the base-rate fallacy is that the limiting factor on IDS performance is the ability to suppress false alarms and not the ability to correctly classify behaviors as intrusive. Using Bayes' Rule [59] shown as Equation 2, the probability of an intrusion given an alarm, $P(I|A)$, can be written as shown in Equation 3.

$$P(A_i|E) = \frac{P(A_i)P(E|A_i)}{\sum_{k=1}^{n} P(A_k)P(E|A_k)} = \frac{P(A)P(E|A)}{P(A)P(E|A) + P(\neg A)P(E|\neg A)} \quad (2)$$

$$P(I|A) = \frac{P(I)P(A|I)}{P(I)P(A|I) + P(\neg I)P(A|\neg I)} \quad (3)$$

Marchette [56] provides an example of the base-rate fallacy. Consider an IDS that monitors 1,000,000 operations where 20 correspond to attacks, or $P(I) = \frac{20}{1{,}000{,}000}$. Also consider a detection rate of 99%, $P(A|I) = 0.99$, and a false alarm rate of 0.1%, $P(A|\neg I) = 0.001$. Using these values gives $P(I|A) = 0.019$, or about 2%. This means that approximately 1 alarm out of 50 operations is an intrusion. With a false alarm rate of 0.01%, $P(I|A) = 0.16$ or 16%. A false alarm rate of 0.001% brings $P(I|A)$ up to 66%.

As the number of false alarms grows, administrators and security officers may become inclined to ignore future alarms. Consequently, it is critical that false alarms be minimized. In order to keep false alarm levels low, it is important that the IDS not introduce elements that trigger an alarm based on benign activity [58]. State tracking can effectively reduce false positives for network-based IDSs. [52]

Relevant metrics from clinical medicine [60][61] applied to IDS environments include:

Sensitivity or True Positive Fraction: Probability that an intrusion is identified when present as given by $\frac{TPI}{TPI+FNI}$.

Specificity: Probability that a detection does not occur when an intrusion is not present as given by $\frac{TNI}{FPI+TNI}$.

False Positive Fraction: $\frac{FPI}{FPI+TNI}$ = 1 - Specificity.

Positive Likelihood Ratio: Ratio between the probability of a alarm in the presence of an intrusion and the probability of an alarm in the absence of an intrusion. It can be written $\frac{Sensitivity}{1-Specificity}$.

Negative Likelihood Ratio: Ratio between the probability of the absence of an alarm in the presence of an intrusion and the probability of the absence of an alarm given the absence of an intrusion. $\frac{1-Sensitivity}{Specificity}$

Positive Predictive Value: Probability that an intrusion occurred when an alarm is present. $\frac{TPI}{TPI+FPI}$

Negative Predictive Value: Probability that an intrusion has not occurred when an alarm is absent. $\frac{TNI}{FNI+TNI}$

Using the formula for conditional probability [59] in Equation 4, it is shown that $P(A|I)$ = sensitivity and that $P(A|\neg I) = 1 -$ specificity in Equations 5 and 6.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{4}$$

$$P(A|I) = \frac{P(A \cap I)}{P(I)} = \frac{TPI}{TPI + FNI} = \text{sensitivity} \tag{5}$$

$$\begin{aligned} P(A|\neg I) = \frac{P(A \cap \neg I)}{P(\neg I)} &= \frac{FPI}{FPI + TNI} = \frac{FPI + TNI - TNI}{FPI + TNI} \\ &= 1 - \frac{TNI}{FPI + TNI} = 1 - \text{specificity} \end{aligned} \tag{6}$$

Summary

Conventional and agent-based IDSs exhibit vulnerabilities resulting from their architectural implementation as can be measured by their TPI, FPI, TNI, FNI, sensitivity, and specificity response to intrusive activity. The reliance on a centralized control structure in tools such as Tripwire and FICA enables failure or tampering tampering at a single point to disable IDS functionality. Frameworks without centralized control are vulnerable to tampering due to static data dispatch. For instance, modification of the Hummingbird misuse database can cause the distributed control modules to produce invalid ID scan results. The CONFIDANT DCDD framework is designed to surmount these limitations as defined in the following chapter.

CONFIDANT OPERATIONAL CONCEPT

Existing conventional and agent-based systems do not emphasize mitigation of insider tampering risks nor are their architectures easily adapted to handle them. However, CONFIDANT specifically addresses the tampering vulnerabilities listed in Table 3 using a distributed control scheme realized with mobile agents [62]. Distribution of both data and control helps to mitigate insider tampering and also eliminates many single points-of-failure present in existing frameworks. By enabling agents to monitor the host file system and compare current and baseline file signature values, CONFIDANT is considered an *agent-based signature-matching* system with *active network components.* Also, since alarms are triggered by changes to monitored files, CONFIDANT signals intrusions in a *behavior-based* [23] manner.

CONFIDANT extends the TACH and FICA frameworks based upon the requirements listed in Table 6. Requirements are listed in order of the progression of IDS processing. Requirements R1, R2, and R3 define the required properties of the environment in which the agents operate. Requirement R4 dictates the security of the agent transactions themselves. These critical requirements involve the integrity of the agent execution, the inter-agent communication, and the processing results maintained during execution. The corresponding operating assumptions OA1 – OA4 ensure that CONFIDANT agents are able to verify the correctness of the individual hosts on the network as well as other agents. Also, file data obtained by agents must be accurate. The listed assumptions are attainable and considered modest given the difficult challenges posed by insider risk. While an insider has full and direct access to any computer system resource, the stated assumptions coupled with use of mobility significantly diminish the ability of any insider to *si-*

Table 6: CONFIDANT Operating Assumptions

Requirement		Operating Assumption	
R1:	File information obtained by agents is legitimate	OA1:	Agents have direct disk access to ensure accurate file validation
R2:	Scan timing is not predictable	OA2:	Agents perform filesystem locking upon arrival
R3:	Agents execute in a protected environment	OA3:	Initial configuration is well-formed and completely installed
R4:	Agent interactions are robust	OA4:	Agent transport and communication occur via SSL to preclude spoofing

multaneously modify every agent in order to compromise file integrity capabilities before an alarm is sounded elsewhere in the network.

The general structure of a CONFIDANT agent is shown in Figure 13. CONFIDANT agents contain executable *behavior code* and a *persistent data repository* to store file integrity data collected while traversing the network. The agent's *itinerary* can be updated dynamically to determine the agent's route in realtime to help preclude tampering. An *agent gateway* resides on each physical processor and provides services required by the agents including *ID*, *mobility*, and *communication management* to mitigate IDS tampering modes as described below.

Towards Insider-Robust Capabilities

Table 7 shows a comparison of related IDSs and illustrates how the CONFIDANT framework differs from existing models. For instance, CONFIDANT employs mobile agents with an interlocked handshaking protocol. The Bernardes-Moreira framework uses mobile agents, but without communication between agents. Several other agent based models dispatch agents, but those agents do not traverse the network domain.

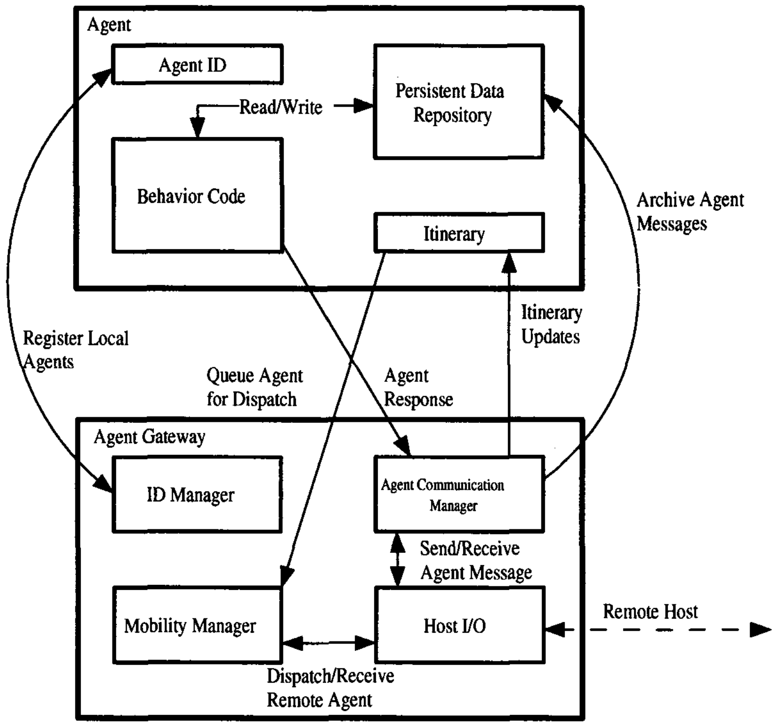

Figure 13: CONFIDANT Agent Structure

Table 7: Comparison of representative File Integrity and Agent-based IDS Approaches

File Integrity Analyzer / IDS	**Execution Model**	**Agent Form**	**Agent-to-Agent Interaction**	**Single Point of Failure**	**Safeguards Against Insider Tampering**
Tripwire, AIDE, Veracity, integrit	Client-Server	N/A	N/A	Yes	No
AAFID	Deployed Agent	Key-Value Pair	None	Yes	No
Bernardes-Moreira	Mobile Agent	Aglet	Create, Halt	Yes	No
TACH	Deployed Agent	Aglet	None	Yes	No
FICA	Deployed Agent	Concordia Java Object	Communicate	Yes	No
CONFIDANT	Mobile Agent	Concordia Java Object	Communicate, Create	No	Yes

Table 8: CONFIDANT Approaches for Tampering Mitigation

Tampering Mode	**Mitigation Approach**	**Description**
Spoonfeeding	Encapsulation	Vulnerable File I/O contained inside agent inside agent
Sugarcoating	Validated transactions	SSL used for messaging and transport
Recanting	Interlocks, scrambling	Agent transactions are interlocked and spatially distributed
Blindfolding	Redundancy, vulnerability seeding	Known exceptions are intentionally inserted to test detection status
Commandeering	Interlocks, scrambling	Agent interactions are interlocked, and spatially and temporally distributed
Soundproofing	Redundancy, interlocks	Alarms at each node with interlocked I/O
Blockading	Pulse-taking	Interlocked file bandwidth monitoring and alert mechanism
Pacing	Pulse-taking	Interlocked CPU throughput monitoring and alert mechanism
Scapegoating	Redundancy	Concurrent tracking via multiple agents
Re-baselining	Distinct Inception	Data dispatched with agent upon configuration
Descoping	Mandatory Obsolescence	Configure only upon initial startup then destroy configuration agents
Value Jamming	Redundancy, scrambling	Agent execution is spatially and temporally scrambled
File Juggling	Redundancy, scrambling	Unpredictable redundant scan scheduling

All of the IDS frameworks described previously have limitations with respect to insider tampering and exhibit a single point-of-failure. Tripwire can be compromised by interfering with the database manager. Compromising a transceiver in AAFID will remove a portion of the network from the control of the IDS. TACH, FICA, and the Bernardes-Moreira framework can be defeated by compromising the agent server. The mobile agent approach in CONFIDANT provides an alternative to these protocols to mitigate specific vulnerabilities. The mitigation strategy developed for CONFIDANT for each tampering mode described previously is listed in Table 8.

The CONFIDANT mobile agent framework consists of an *agent gateway* on each monitored host, four agent *behaviors*, and agent *interaction* operating in three *echelons.* The agent gateway provides the interface between the agents and services on each host on the network as well as the communication mechanism for the agents to travel over the network. The behaviors are designed around specific lifespan and operation cycles to perform specific functions within the agent framework. Agent interactions include communication between individual agents as well as between agent groups.

Agent Gateway

The CONFIDANT *agent gateway* provides the interface between the host services and the agents. It also maintains network communication channels for agent dispatch to remote nodes and agent interaction. In order to allow agents to travel to remote nodes, a gateway must be executing on every monitored host in the network. Each gateway, $G = \{g_1, \ldots, g_n\}$, is a monitored network node.

Each mobile agent gateway:

- creates and disposes of application agents,
- monitors local agents executing on a host,
- receives reports generated by remote agents, and
- forwards reports generated by local agents to remote gateways.

Tampering at the gateway level remains a potential vulnerability. To maintain IDS capability, the gateway needs to be protected from tampering and disabling. This is addressed by the assumptions listed in Table 6, and is a topic of current research [63] [64] [65]. Disabling a CONFIDANT gateway results in alarm notification when either an agent attempts to travel to the gateway or an attempt is

Table 9: CONFIDANT Agent Behaviors

Behavior	Life span	Execution Cycles Per Gateway Visit
Probe	One Execution Cycle	Exactly One
Sensor	Infinite	Exactly One
Beacon	Infinite	Multiple Based On Time Intervals
Watchdog	Infinite	Multiple Based On Target Criteria

made to communicate with an agent that was residing on the gateway when it was disabled.

Autonomous Behaviors

The mobile agent framework makes use of four distinct types of agents, each defined for a specific *life span* and *execution cycle.* An agent life span is the duration from initial dispatch until execution is complete according to its *itinerary.* An agent execution cycle includes agent-specific processing and related messaging. For instance, the execution cycle for an agent that computes file MD5 values is simply reading a file and calculating the MD5 which occurs once per gateway visit. The execution cycle for an agent that correlates MD5 values is to receive and process a message from a MD5 computation agent. Based on messages received, multiple event cycles can occur on a single gateway visit.

Every CONFIDANT agent behavior extends the general agent structure illustrated in Figure 13. Individual behaviors are defined in the following subsections and summarized in Table 9. Each agent has a specific behavior based on lifespan and execution cycles per gateway visit. All agents regardless of behavior, perform the same dispatch and communication functions described below.

Probe

The *Probe* agent behavior is the most basic type of CONFIDANT agent. Its life span is only a single execution cycle. A Probe agent is dispatched to the destination gateway, an arrival confirmation message is sent, and the agent is terminated once execution is complete. Probe agents are primarily used for sounding alarms and sending messages to other remote gateways.

Sensor

Sensor agents, like Probe agents, have a single execution cycle per gateway visit, but differ in that their life span is perpetual. One role of Sensor agents is to inspect for changes to the network topology, such as hosts added or removed from the network.

Beacon

Beacon agents have an infinite lifespan and can have multiple execution cycles per gateway visit. As the name implies, beacon execution cycles occur at regular time intervals. Since Beacon agents report on regular intervals, several cycles may occur while the agent resides on a single gateway.

Watchdog

As is the case with Beacon agents, *Watchdog* agents have an infinite life span and multiple execution cycles per gateway visit. Unlike Beacon agents, however, Watchdog agents perform execution cycles based on a certain target criteria. For instance, a Watchdog agent is used to monitor agent filesystem scan results. When a discrepancy is encountered a Watchdog agent dispatches Probe agents to specific nodes to sound the alarm.

Agent Echelons

CONFIDANT agents operate in distinct echelons as shown in Figure 14. In simple domains with limited overhead, functionality can be encompassed by Watchdog agents to perform filesystem scans and Probe agents to send alarm notification to remote gateways. As network complexity and overhead increases, functionality can be distributed into distinct functional components as described below.

As shown in Figure 14, the lowest echelon is responsible for *surveillance.* In this level, *Sensor* agents are dispatched in order to obtaining file signatures and information regarding network topology. State detection sensor agents monitor the network for changes in gateway status. For instance, if the power is removed from a particular host and the gateway becomes unavailable, state detection agents will signal the network topology agents to make appropriate itinerary updates with other agents. File inspection sensor agents are responsible for visiting agent gateways and obtaining file signatures to transmit to the data correlation module.

The middle echelon is the *Control level* which provides updates for agent itineraries as well as performs result collection and correlation. The network topology module is responsible for obtaining network status information from the sensor echelon and relaying appropriate updates to the file inspection, data correlation, and alarm dispatch modules. The data correlation module receives MD5 signatures from the file inspection module at the sensor echelon. Here data is collected and analyzed to provide error information to the alarm dispatch module. Both the network topology and data correlation modules utilize the Beacon agent behavior.

The final echelon is the *Response level* and consists of the alarm dispatch module. Watchdog agents are used to collect and distribute alert notification messages from the control level to remote gateways. While alarm messages can be sent

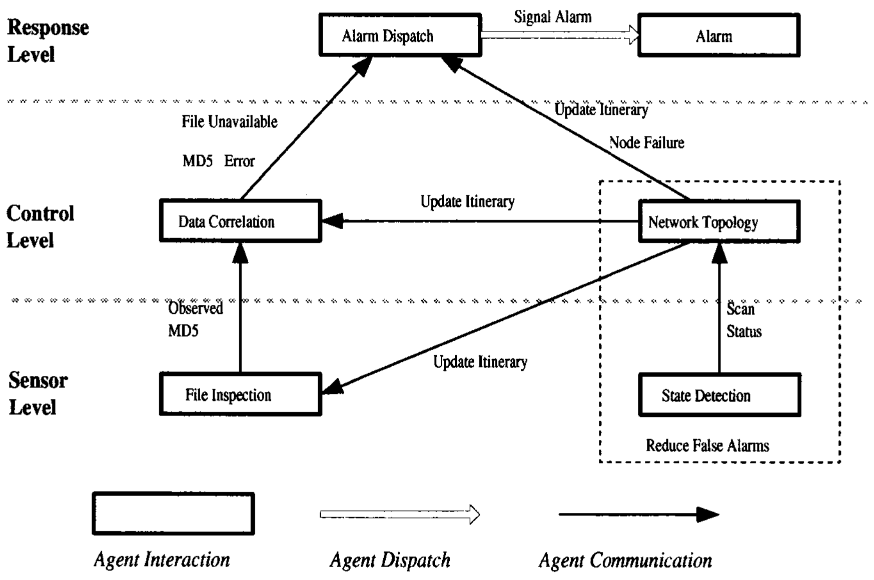

Figure 14: CONFIDANT Echelons

to remote agents, gateways that are not currently hosting agents are unable to receive messages directly. In order to overcome this limitation, Probe agents are dispatched by the alarm dispatch module to transport appropriate alarms. Probe agents travel to the remote gateway, reply with an acknowledgment to confirm arrival, and perform alarm notification routines. Encapsulation of alarm messages within an agent enables alarms to be signaled on any gateway on the monitored network.

Reducing the number of false alarms is of particular importance as discussed previously. The Network Topology and State Detection modules in Figure 14 are designed specifically to reduce redundant alarms. Consider a subnet of gateways that are subject to frequent resetting, as in a student computer lab. Once gateway termination is detected, agents signal alarms to warn of tampering. State Detection agents relay gateway status to Network Topology agents. Messages are subsequently transmitted to other agents in order to provide itinerary update data. If potential tampering is detected and appropriate alarms triggered, agents can omit future scans and reduce the number of alarms.

Agent Interaction

CONFIDANT agents operating across the defined echelons are in frequent communication with each other. Groups of similar agents are divided into *committees* in order to provide coverage of a network domain. A single committee C^i is defined as $C^i = \{a_1^i, \ldots, a_n^i\}$ where i is the committee index and n is the number of agents in committee C^i. The set of all committees within a monitored network is defined as $C = \{C^1, \ldots, C^m\} = \bigcup_{i \in m} C^i$ where m is the total number of committees. Adjacent committees share common agents in order to enhance scalability and adapt to physical network layout. Due to the overlapping nature of adjacent committees, $\bigcap_{i \in m} C^i \neq \emptyset$.

Agents Independent of Committee Interaction

Some agents, specifically Probe agents, are not members of a committee and are considered independent. Handshaking interlocks provide assurance that messages sent to remote agents are received. Independent agents provide robust communication, particularly alarm messages, to remote gateways that are not hosting agents at the time the message is generated.

Independent agents are used to provide robust communication to monitored network nodes that are not currently hosting agents. The set of independent agents, $I = \{a_1^I, \ldots, a_n^I\}$, includes all CONFIDANT agents that are not members of any committee. An agent j that is a member of committees i and k is denoted $a_j^{i,k}$, and has the following properties

- $a_j^{i,k} \in C^i$,
- $a_j^{i,k} \in C^k$, and
- $a_j^{i,k} \notin I$.

Thus, the set of all CONFIDANT agents, A, is defined as $A = I + \bigcup_{i \in m} C^i$.

Interaction Within Committees

An agent *committee* is a group of related agents that operate across a division of the network domain and communicate via broadcast messages to other members. Members within a committee are bound to a physical subset, $g \subset G$, of the monitored network. Table 10 lists eight agent event types. These events are used to provide interlocked transactions so that agents are aware of both status and location of other committee members. Interlocking provides robust agent communication in order to ensure messages are received by remote committee members across the network domain. The ability to monitor remote agents is a requirement of the CONFIDANT design goals.

Figure 15 illustrates communication between agents within a committee. Three agents are shown. Two agents a_1^1 and a_2^1 from committee C^1 and a_1^2 from C^2. Agents can receive messages only from other committee members. Consequently, agent a_2^1 can receive messages from agent a_1^1, but not from agent a_1^2.

Table 10: CONFIDANT Agent Event Types

Event Name	**Description**
`AgentArrived`	Sent when a dispatched agent arrives at a remote gateway
`AgentArrivedACK`	Sent to a recently dispatched agent upon receiving arrival notification
`AgentTravelRequest`	Sent by an agent to Committee members when prepared for dispatch
`AgentTravelProceed`	Response from Committee members when dispatch is acknowledged
`MD5OK`	Sent upon valid MD5 computation
`MD5Error`	Sent upon modified MD5 computation
`MessageACK`	Sent to confirm message receipt
`AgentUnavailable`	Sent to Committee members when a member does not respond to communication
`HostUnavailable`	Sent to Committee members when travel to a gateway is prohibited

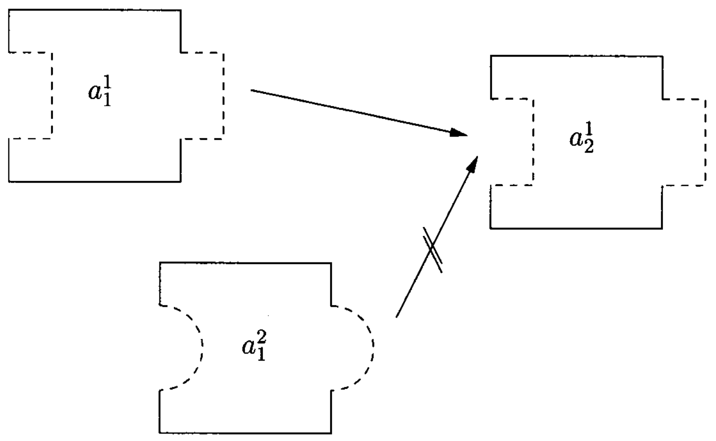

Figure 15: Agent Communication Within A Committee

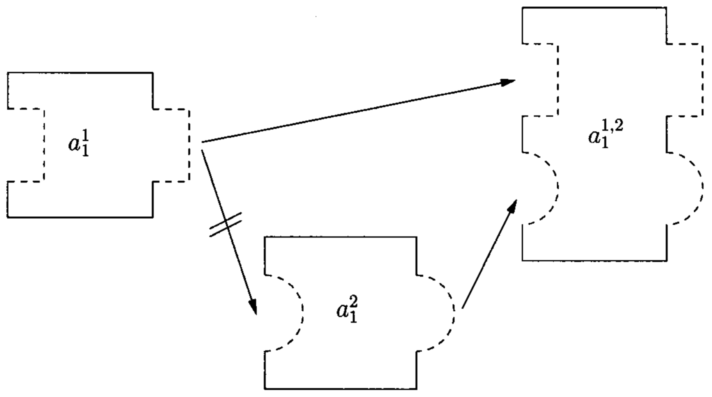

Figure 16: Agent Communication Between Overlapping Committees

Interaction Between Committees

Overlapping committees are used in order to take advantage of physical network domain configuration and enhance scalability. Agents within a committee maintain peer-to-peer connectivity within their group as illustrated previously. The common nodes between adjacent committees enable monitoring of different portions of the network.

Figure 16 illustrates communication between agents where one is a member of two overlapping committees. In this example, agent a_1^1 is a member of committee C^1 and agent a_1^2 is a member of committee C^2. Agent $a_1^{1,2}$ is a member of both C^1 and C^2 and consequently enables communication between both committees. Here agent a_1^1 is unable to communicate directly to agent a_1^2, but both are able to communicate with members of the other committee due to the overlapping nature of agent $a_1^{1,2}$.

Agent Dispatch and Communication

In order to describe dispatch and communication between agents, dispatch and communication terms are defined in Table 11 and functions are defined in Table 12. For example, a dispatch operation is denoted by:

$$a_j^i.dispatch(g_k) < delay, \Delta t_d > .$$

Consider a Watchdog agent a_4^6 in the Alarm level creating a Probe agent immediately to dispatch to gateway g_2 in order to warn of an file modification. This operation is denoted:

$$a_4^6.spawn(probe,\ MD5Error,\ g_2) < 0, \Delta t_d > .$$

The *operation* delay of 0 signifies that the operation is to take place immediately. The maximum delay allowed for agent dispatch is defined as Δt_d. All Δt terms in Table 12 define a waiting period for operation confirmation. In the previous example, if an arrival acknowledgment message is not received from the Probe agent created by the spawn operation within Δt_d, an error has occurred and is handled accordingly.

Next, consider an agent a_2^2 preparing to travel to a remote gateway by sending a travel request message to other members of committee C^2 after a 4 second delay. The corresponding operation is:

$$a_2^2.sendmsg(C^2,\ AgentTravelRequest) < 4,\ \Delta t_s > .$$

Table 11: CONFIDANT Agent Dispatch and Communication Terms

Term	Definition
`listener`	The listening agent or committee.
`msg`	The message transmitted between agents.
`delay`	The time between when a particular operation is specified and when it is performed. Delay=0 means to perform the operation immediately.
Δt_s	The waiting period from when a message is sent until a confirmation is received.
Δt_r	The time allowed to wait for an expected message to be received.
Δt_d	The time allowed between agent dispatch and the related arrival message.

Table 12: CONFIDANT Agent Dispatch and Communication Functions

Description	Definition
Creation of a new agent	spawn(b_i, msg, g_i)<delay, Δt_d >
Agent travel to remote gateway	dispatch(g_i)<delay, Δt_d >
Transmitting a message with no expected response	sendmsg(listener, msg)<delay>
Transmitting a message and waiting for a response	sendmsg(listener, msg)<delay, Δt_s >
Listening for a message	receivemsg(sending agent, msg)< Δt_r >

An example of agent dispatch and communication illustrating use of these definitions and functions is provided in below.

CONFIDANT Handshaking Scenarios

At time $t < 0$, the monitored network is not connected to the outside network. All gateways, $g_i \in G$, are operating and all agents in committees, $C^j \in C$, are executing. The network is in a safe state and CONFIDANT is operating on all monitored nodes. Sensor agents are dispatched in the sensor/inspection layer to perform file inspection and state detection operations. Beacon agents are deployed in the control layer to correlate responses from the sensor level and provide alert data to the response echelon. Watchdog agents in the response echelon await alert notification from the control level agents.

At time $t = 0$, network monitoring commences. File Sensor agents continue to traverse the network updating MD5 values for correlation by beacon agents in the control layer. While message digests are being collected and correlated, State Detection agents traverse the network scanning for any changes in resource availability. If a particular gateway is subject to frequent downtime, the itinerary of other agents are updated in order to prevent frequent attempts to visit an unavailable host. State Detection agents investigate previously unavailable nodes in order to update the Network Topology module in the control layer that resources are again available and to update agent itineraries accordingly.

Agent dispatch and communication is illustrated in Figure 17. In this example agent a_1^1 is dispatched to host g_2. Committee C^1 agents are notified and respond during a_1^1 dispatch. Dispatch and communication use handshaking with acknowledgment to ensure agents are aware of other members of their committee. Each agent a_i^1 first sends an intent to travel message and waits for confirmation prior to dispatch. Upon arrival on the remote gateway, an arrival message is sent to all committee members $a_{j \neq i}^1$. The committee members then respond with an arrival acknowledgment message. Dispatch is then complete.

Handshaking is essential to guarantee that CONFIDANT meets Goal-1. Consider agents not maintaining communication with other members of the group. If the agents are allowed to occupy the same host at the same time, failure of that gateway would be a single point-of-failure. Distributing redundant mobile agents in multiple interactions and levels ensures that no single agent, behavior, interaction, or level is completely responsible for IDS operation. Consider a single

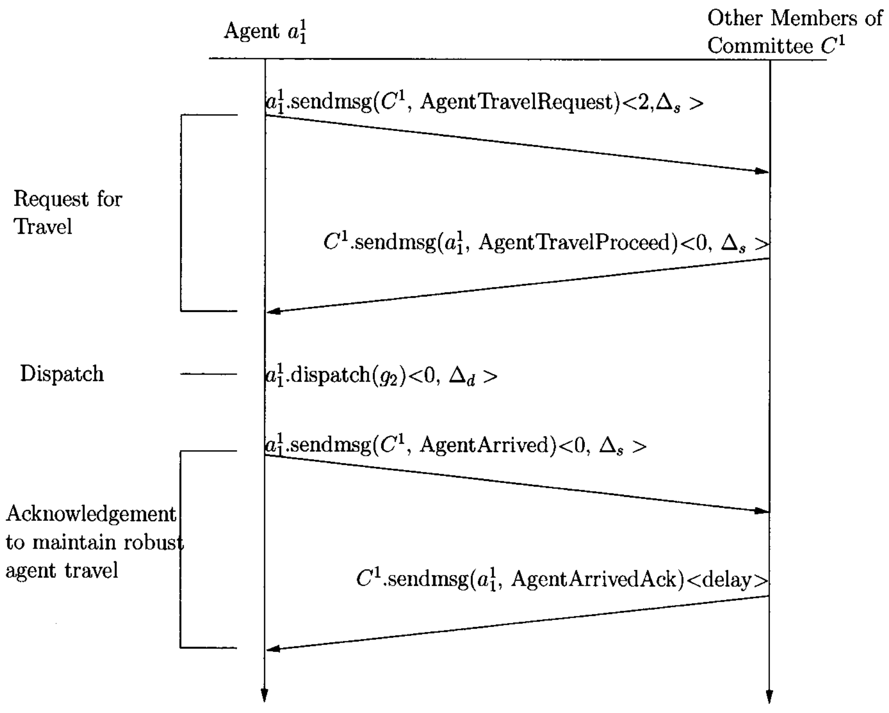
Agent a_1^1
Other Members of Committee C^1
a_1^1.sendmsg(C^1, AgentTravelRequest)<2,Δ_s >
Request for Travel
C^1.sendmsg(a_1^1, AgentTravelProceed)<0, Δ_s >
Dispatch
a_1^1.dispatch(g_2)<0, Δ_d >
a_1^1.sendmsg(C^1, AgentArrived)<0, Δ_s >
Acknowledgement to maintain robust agent travel
C^1.sendmsg(a_1^1, AgentArrivedAck)<delay>

Figure 17: Dispatch and Communication Example

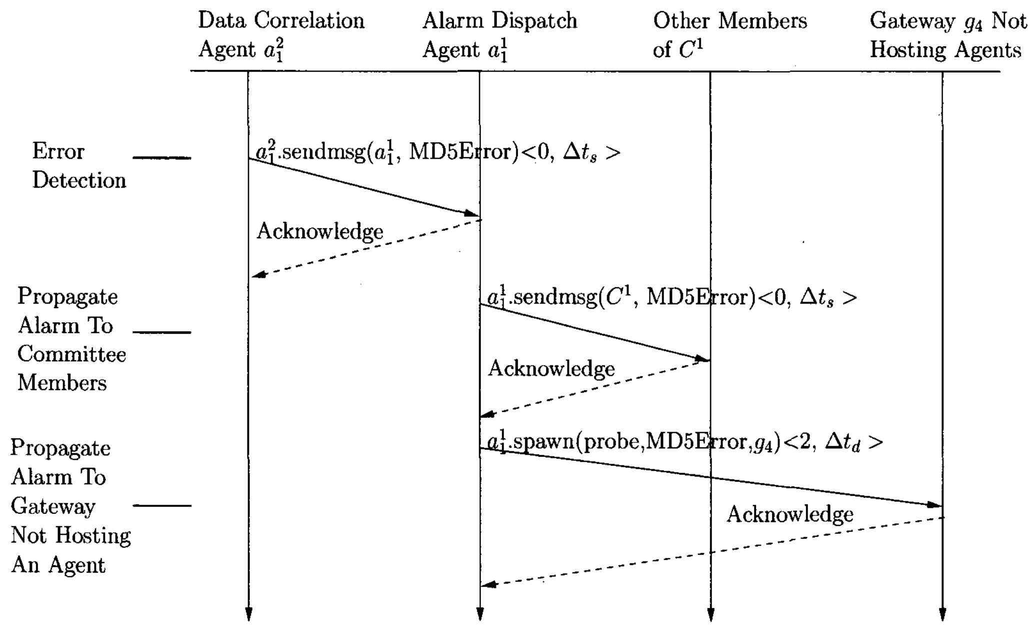
Data Correlation Agent a_1^2
Alarm Dispatch Agent a_1^1
Other Members of C^1
Gateway g_4 Not Hosting Agents
Error Detection
a_1^2.sendmsg(a_1^1, MD5Error)<0, Δt_s >
Acknowledge
Propagate Alarm To Committee Members
a_1^1.sendmsg(C^1, MD5Error)<0, Δt_s >
Acknowledge
Propagate Alarm To Gateway Not Hosting An Agent
a_1^1.spawn(probe,MD5Error,g_4)<2, Δt_d >
Acknowledge

Figure 18: Propagation of Alarm Notification

committee of n agents. Disabling any $x < n$ agents results in $n - x$ agents creating alarm reports stating that communication has been disrupted. Consequently, a successful CONFIDANT adversary would need to tamper with every node in the monitored network simultaneously. Additional details are provided in the evaluation of Goal-1 below. In order to provide redundancy, at least $n = 2$ agents of a specified behavior must be dispatched at each level and interaction. The optimal number of agents dispatched at any level or interaction is currently being assessed as it depends on network characteristics such as topology, latency, and size. This redundancy helps ensure that agent termination or node failure does not compromise CONFIDANT operation.

If at any point a resource is unavailable or an observed file MD5 value differs from the baseline, Beacon agents in the control layer send alert notification to listening Watchdog agents in order to notify administrators of the modification. Figure 18 illustrates detection of a file modification. A Data Correlation agent, a_1^2 determines that the obtained MD5 value does not match the expected result. An MD5Error message is sent to an Alarm Dispatch agent to propagate the error to other nodes. Each message has a subsequent acknowledgment to provide interlocking. The error message is relayed directly to other members of committee C^1. In this example, gateway g_4 is not hosting agents and therefore can't receive messages directly. Agent a_1^1 spawns a probe agent after a two-second delay in order to transport alarm notification to the remote gateway. These interactions have been implemented on the Concordia mobile agent framework.

CONFIDANT Detailed Design

The primary function of CONFIDANT is to perform filesystem scans. Agent dataflow is illustrated in Figure 19 using the notation described in [66]. Integrity scans are performed by obtaining file contents, computing the file MD5 hash value, and comparing that result to the internal baseline data. Upon scan completion, the result is sent to other committee members to corroborate the result. Next, the agent travel operation commences. Prior to dispatch, the agent will parse the internal list of monitored gateways. Once a gateway is selected, the agent will send a travel request to committee members in order to maintain communication upon arrival at the remote gateway. When the agent arrives at the destination gateway, arrival notification is sent to committee members and filesystem scans resume.

Pseudocode for the scan operation is provided in Figure 20. The `computeMD5` function is performed to obtain the MD5 hash value of the monitored file. A `MD5OK` event is created if the result matches the baseline. If the file has been

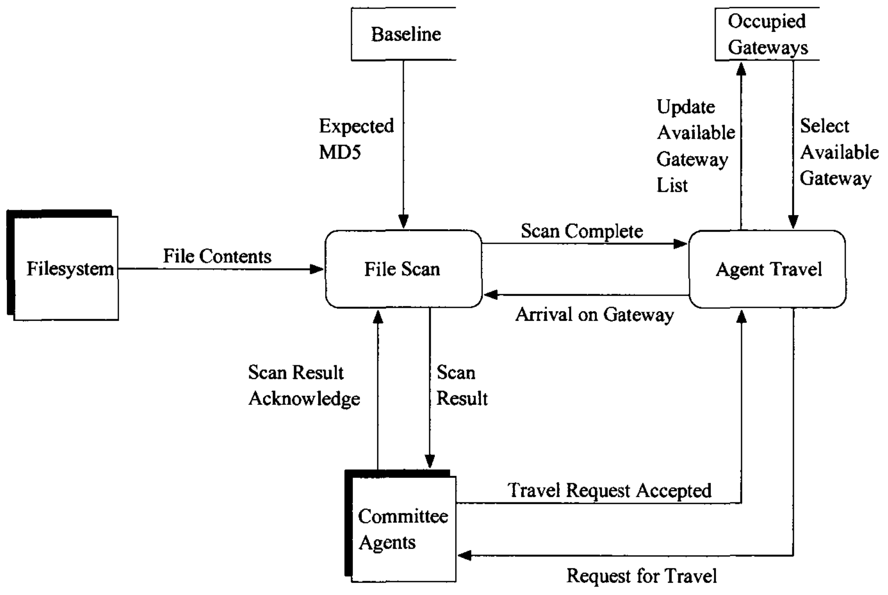

Figure 19: CONFIDANT Dataflow Diagram

modified, a `MD5Error` event is created. The event generated as a result of the scan is then distributed to other committee members. The name of each committee agent is stored locally in an array `ca[1..n]` along with the name of the gateway on which the agent is operating. A `foreach` loop is used to send the scan result to each committee member. The agent will then `sleep` for Δt_s to allow committee member responses to arrive. Responses are handled asynchronously and stored in an array. The responses are then processed to ensure the message was successfully conveyed.

In order to maintain agent interlocking, three communication sequences occur between distributed committee members as illustrated in Figure 21, as based upon the notational conventions in [66]. The first sequence occurs upon file scan completion. Once the scan is complete, results are sent to committee members. The remaining sequences enable committee agents to maintain robust communication

```
gateways = g[1..m];
committee agents = ca[1..n];
responses = r[1..n];

scanresult.setValue(computeMD5(filename));
if (scanresult.getValue() == baseline) {
     result = new ConfidantEvent(MD5OK, scanresult.getValue());
} else {
     result = new ConfidantEvent(MD5Error, scanresult.getValue());
}
sendScanResult(result) {
     foreach i in ca[] {
          if (ca[i].g != null) {
          sendmsg(ca[i].g, result)
          }
     }
     sleep(delta_ts);
     processResponse();
}
```

Figure 20: CONFIDANT Pseudocode for File Scan Operation

while traveling between remote gateways. Prior to dispatch, an agent will select an available gateway and notify committee members of intent to travel. Committee members respond in order to confirm that communication will be maintained upon arrival at the destination gateway. Once confirmation is received, dispatch commences. If the remote gateway is unavailable, an alarm is triggered, a new destination gateway is selected, and the interlocking communication process repeats. Upon arrival at the remote gateway, arrival notification is sent to committee members in order to ensure that future messages are transmitted to the correct gateway. An alarm may result from any of these three sequence if message acknowledgment is not received within Δt_s or if agent dispatch is not successful within Δt_d.

Pseudocode for the agent dispatch communication sequence is provided in Figure 22. First, `selectGateway` is called to determine the agent dispatch destination. A new `AgentTravelRequest` event is created and sent to committee agents as described previously. The notation `ca[i].g` represents the gateway on which the agent `ca[i]` is operating. A value of `null` indicates that travel or communication for the agent `ca[i]` has failed. Once responses are processed, the agent is dispatched to the remote gateway. Upon arrival at the destination, `sendArrivalNotification` is called to inform committee agents travel success.

Implementation on Concordia Prototyping Platform

Concordia [67] is a framework for the development and management of mobile agent applications which extend to any device supporting Java. A *Concordia System* is made up of numerous components, each of which are integrated together as listed in Table 13. The *Concordia Server* is the major component, inside which the various *Concordia Managers* reside. A Concordia System must include a *Java*

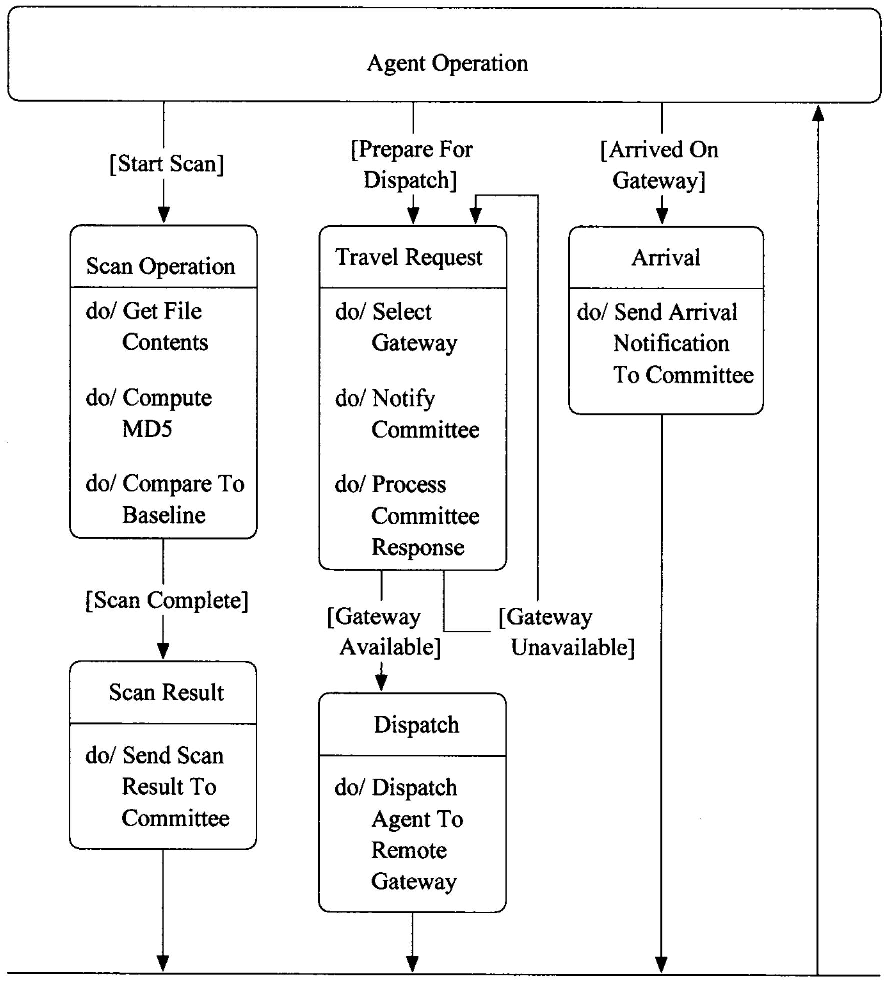

Figure 21: CONFIDANT Operation State Diagram

```
gateways = g[1..n];
committee agents = ca[1..n];
responses = r[1..n];

gateway = selectGateway();
destination = new ConfidantEvent(AgentTravelRequest, gateway);
sendTravelRequest(destination) {
     foreach i in ca[] {
          if (ca[i].g != null) {
          sendmsg(ca[i].g, destination)
          }
     }
     sleep(delta_ts);
     processResponse();
     dispatch(destination);
     sendArrivalNotification();
}
```

Figure 22: CONFIDANT Pseudocode for Agent Dispatch

Virtual Machine (VM), a *Concordia Server*, and at least one mobile agent on at least one network node.

It is common for a Concordia Server to execute on each node of the network. An agent must invoke server methods to initiate transfer in order to travel between nodes. Each agent maintains an *itinerary* used by the server to determine the order in which nodes are to be visited and what operation is to be performed at the visited nodes. The server contacts the destination node to begin transfer. In order to provide a reliable guarantee of transfer, the agent is stored persistently before being acknowledged. The agent is queued for execution on the receiving node upon completion of the transfer. When execution begins, the agent is restarted on the new node according to the method specified in its itinerary. When execution is complete, the server again inspects the itinerary to determine the next destination.

Table 13: Concordia Components

Concordia Server	The complete Concordia component running in the network comprised of manager components
Agent Manager	Communication infrastructure that allows agents to be transmitted and also manages the life cycle of the agent
Administrator	Manages all of the services provided by Concordia,
Security Manager	Responsible for identifying users, authenticating agents, protecting server resources and ensuring the security and integrity of agents
Persistence Manager	Maintains the state of agents in transit around the network.
Event Manager	Handles registration, posting, and notification of events to and from agents
Queue Manager	Responsible for the scheduling and possibly retrying the movement of agents between servers
Directory Manager	Provides naming service in the agent framework
Service Bridge	Interface from agents to the services available at various machines in the network
Agent Tools Library	Provides all the classes needed to develop Concordia Mobile Agents

Mitigation Techniques

Spoofing-based Tampering

Spoofing occurs when counterfeit data is transmitted to the recipient. Three spoofing attacks are considered. The first is *Spoonfeeding* sensor information at TP_{FS} that is not present in the target file. As listed in Table 8 this is mitigated in CONFIDANT by encapsulation of the interface between the agents and native services on the host. CONFIDANT's agent gateway enables the agents to access the host filesystem directly to mitigate this exposure as described previously.

The second attack considered is *Sugarcoating* of unfavorable reports. This is mitigated in CONFIDANT by using SSL encryption for messaging and transport in order to validate all agent communication and transfer. Agents will also perform integrity verification on the agent gateway to determine if tampering has occurred at the gateway level.

The third spoofing-based attack considered is *Recanting* of alert notification. CONFIDANT mitigates Recanting by enforcing transaction interlocks between agents. Agents must remain in constant communication. If agent communication is interrupted and handshaking is not maintained, then a suspicious activity has occurred which activates an alert.

Termination-based Tampering

Disabling an IDS sensor is called *Blindfolding*. This is mitigated in CONFIDANT by enabling multiple agents to perform similar tasks. Agent a_i^1 remains in communication with all agents $a_{j \neq i}^1$ in committee C^1. If Δt_r is exceeded then agent a_i^1 is determined to be missing by not maintaining an appropriate communication channel, or if an agent gateway cannot be contacted then an alert is initiated. Furthermore, one agent can alter a file in order to verify that other cooperating agents are correctly identifying file modifications.

Overriding of IDS decision-making operations, or *Commandeering*, is mitigated in CONFIDANT by distributing all decision-making responsibilities in the form of redundant mobile agents. Multiple agents in each committee C^i perform the same functionality to mitigate tampering at any single point. Transactions between agents within a committee C^i, as well as between overlapping committees, are interlocked and are also both spatially and temporally distributed.

Soundproofing an IDS framework involves muting the alarm to preclude end-user notification. This is mitigated in CONFIDANT by providing communication with multiple agents and by interlocking I/O via the agent gateway. If messages from a remote agent are expected and not received upon the expiration of the Δt_r window, an alert is initiated. Each gateway g_i provides agents with direct access to system resources so that alert notification is reliably transmitted to the security administrator.

Sidetracking-based Tampering

Some frameworks are subject to *Blockading*, or isolating a sensor from needed access to a component or data. CONFIDANT mitigates Blockading by distributing the investigating and decision-making responsibilities. If agent throughput is limited and agents are not able to access either a network node or a service on the host within a specified time, the node is considered suspect and an alert is initiated. CCSD, CCDD, and DCSD architectures are particularly vulnerable as Blockading can be focused on a single point either at the network interface, TP_N, or at the host process level, TP_{PT}.

Altering execution rates, or *Pacing*, is mitigated in CONFIDANT by redundancy of agents in committees C^i. Multiple committee agents traverse the network to analyze files on remote computer systems. Agent actions are based on internal timers defined by individual agents and not on the time of day, thus mitigating tampering at TP_{SC}. All agents a^i_j must provide status messages to cooperating agents in committee C^i prior to expiration of Δt_r, or else tampering is suspected.

An example of *Scapegoating* is triggering an alarm as a decoy in order to hide an actual attack. This is mitigated in CONFIDANT by enabling committee C^i agents to pursue each simultaneous alert independently so that multiple alerts can be processed concurrently.

Internal Data Tampering

File integrity tools create an initial baseline reference for future file verification. *Retroactive Baselining* modifies the reference values thus corrupting the baseline at TP_{ID}. This is mitigated in CONFIDANT by maintaining baseline data within each agent responsible for file integrity verification. When an agent a_1^1 computes a cryptographic digest for a file, the result is compared to internal baseline data encapsulated within multiple mobile agents and transmitted via events to agents in C^1. If the internal data is modified, agent redundancy enables file verification to be performed by other agents.

IDS control components are subject to *Descoping* by tampering with the initial policy configuration data. As with Retroactive Baselining, this occurs at TP_{ID}. This is mitigated in CONFIDANT by including policy information within each agent. Agent a_1^1 contains the list of monitored files as defined prior to initial dispatch. The internal data is not modified during network traversal. If the policy information for a critical file is somehow maliciously altered within a_1^1, redundancy of agents in committee C^1 ensures that particular file will be inspected by other agents. Also, if policy data has been maliciously altered, the absence of messages provided by a_1^1 to other members of C^1 reflect that a monitored file was not scanned. This is detected by agents in C^1 as tampering.

Value Jamming occurs at the alarm level and involves interference with a malicious high-priority process altering the contents of memory. This is mitigated in CONFIDANT by enforcing each committee agent to be responsible for maintaining file status information. Multiple agents reside simultaneously on a node at any given time, so there is no single memory location that serves as a vulnerable status flag. Memory locations used for status information can also vary each time an agent a_j^i visits a node due to occupying a different memory location. Since status flag memory locations can be both spatially and temporally distributed for each agent visitation, CONFIDANT is less vulnerable to tampering by jamming.

Selective Deception

In order for a framework to be subject to tampering at TP_{FS} by *File Juggling*, an adversary must be able to predict that a file integrity scan will occur at time t_{scan} as to perform undetected file system modifications. Selective Deception is mitigated in CONFIDANT by enabling multiple redundant agents a_j^i operating in committee C^i to have a unique itinerary and scheduling parameters. Agent visitation does not occur at regular intervals. It is not required for an individual CONFIDANT agent to visit every node, but coverage of all nodes is guaranteed by the use of multiple agents each with an independent itinerary.

CONFIDANT EVALUATION

The ultimate goal of an IDS is to alert administrators and security personnel to computer system tampering. An IDS is a two-category classifier where the presence or absence of an intrusion results in a corresponding presence or absence of an alarm. Test cases were developed to evaluate the CONFIDANT intrusion response against Goal-1 and Goal-2. A weighted response analysis was then used to numerically compare Tripwire, AIDE, and CONFIDANT.

Testing Objectives and Environment

CONFIDANT testing is designed around four objectives:

1. qualitative performance accuracy in the absence of tampering,
2. agent network performance in the presence of increasing load,
3. Goal-1 specific evaluation in the presence of gateway failure, and
4. Goal-2 specific evaluation of insider tampering resistance to defined tampering modes.

The CONFIDANT evaluation network is composed of four physical hosts running a combination of Linux and Windows connected to a single switch. Quantitative tests involve a single committee C^1 of Watchdog agents a_i^1 traversing the network domain. Functionality of all echelons defined previously is encompassed by up to two committees.

A single file named `confidant.test` is used for MD5 verification testing. Two files generated by `/dev/urandom` on a default Debian Linux install exist to provide

Table 14: Numerical measures for the file integrity problem

Numerical Measure	**Description**
True Positive Intrusion (TPI)	Modification of a monitored file followed by an appropriate alarm
False Positive Intrusion (FPI)	The presence of an alarm when no file modification has occurred
True Negative Intrusion (TNI)	The absence of both a file modification and an alarm
False Negative Intrusion (FNI)	A file modification without an associated alarm
Sensitivity (Sen)	Probability that a file modification is identified when present ($\frac{TPI}{TPI+FNI}$)
Specificity (Spec)	Probability that an alarm is not sounded when a file modification is not present ($\frac{TNI}{FPI+TNI}$)

both a valid version and a modified version of the file under investigation. The valid version, `confidant-valid.test`, represents an unmodified file while the alternate version, `confidant-modified.test`, represents a file that has been tampered with. The MD5 value for valid file is `50a1f6525d4ae14ecabfdb2fe8f03e0f` while the MD5 for the modified data is `0a362a35a204fcc74ab6296f3ff1bb13`. The appropriate file is renamed to `confidant.test` as required for testing described in the following sections.

For file integrity analyzers, a file modification is considered an intrusion, so a True Positive Intrusion (TPI) result occurs when a file modification is accurately detected, while a False Positive Intrusion (FPI) occurs when an alarm is sounded in the absence of a file change. Similarly the absence of an alarm following a file modification is a False Negative Intrusion (FNI), while the absence of both a file modification and alarm is a True Negative Intrusion (TNI). These definitions along with others introduced previously are listed in Table 14.

An alarm generated in response to an authorized file modification is a false positive intrusion. Leveraging the fact that any alarm might potentially provide useful intrusion information, every file modification encountered by CONFIDANT agents that is not permitted by the distributed policy data is reported and considered a

TPI. An alarm generated by an agent a_j^i upon expiration of Δt_r due to network outages may be considered a FPI if it is the result of benign activity. The presence of an alarm generated by such activity is considered to be a TPI, as network or gateway failure may be an indication of malicious intent where an insider tries to circumvent CONFIDANT agent interactions. In this case, alarms are generated in order to notify administrators of the unavailability of network resources.

Tripwire and AIDE are tested in addition to CONFIDANT for comparison. The versions tested are Tripwire 2.3.12 and AIDE 0.10 as available in the default Debian package repository. Both are configured to verify the integrity of a single file, `confidant.test`, as described previously. Also, both are executed using the `cron` daemon. Tripwire and AIDE, using the default configuration, perform integrity scans once daily. This is consistent with the NIST recommendation for file integrity scan intervals [68].

Evaluation for Goal-1

CONFIDANT is designed to mitigate any single points of failure by employing mobile agents to realize a DCDD architecture. Currently existing IDS frameworks described previously exhibit a single point-of-failure by which IDS functionality is defeated if, in the worst case, a single network node is tampered with. The AAFID architecture documentation even states that a disadvantage of the framework is the existence of a single point-of-failure [69].

The agent interaction serves in part to ensure that all agents within a committee C^i do not reside on the same host. If at any time committee agents were to reside on the same physical host and that particular gateway or associated network transport mechanisms were to fail, then CONFIDANT would generate a FNI. For the set A of agents and the set G of gateways, agent interlocking ensures that multiple agents do not occupy the same gateway when $|A| < |G|$. When $|A| \geq |G|$, agents are allowed to travel to a gateway currently hosting agents if all permissible gateways are occupied.

Termination of a gateway is detected by remote agents. If a gateway is hosting an agent when terminated, the agent is also terminated thereby disabling some IDS capability. Disabling of all agents causes a false negative response as CONFIDANT is no longer executing. The following discussion assumes that only gateways hosting agents are terminated as this is the worst case scenario. Termination of $n = |A|$ gateways results in termination of all agents and disabling of CONFIDANT. For the case in which each gateway hosts a single agent, consider a committee C^1 of agents operating in a network of gateways where $|A| = |G|$. Upon agent dispatch, one agent will travel to a node currently hosting an agent, thus resulting in one gateway hosting 2 agents, another hosting 0 agents, and the remaining $|G| - 2$ nodes each hosting a single agent. With communication interlocking ensured such that agents do not congregate to a limited subset of network nodes, $n = |A| - 1$ nodes must be tampered with within time Δt_r in order to terminate CONFIDANT monitoring capabilities. In order for an attacker to force a false negative result from CONFIDANT operation, $n \geq |A| - 1$ gateways must be terminated within time Δt_r.

Assume by contradiction that disabling $n < |A| - 1$ gateways will disable CONFIDANT operation. In this case, a minimum of $(|A| - 1) - n$ agents remain as some gateways may not have been hosting agents at the time of termination. The remaining executing agents will continue IDS operation. In the worst case,

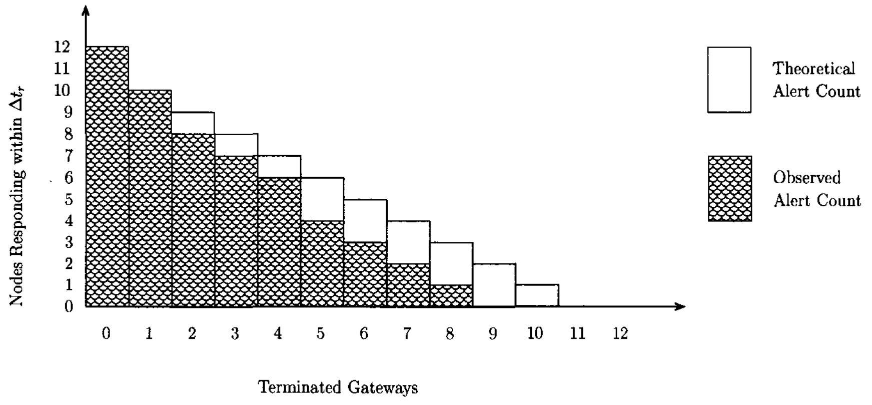

Figure 23: Agent Response in the Presence of Gateway Failure

no remote agents or gateways are able to be contacted; however, local agents still continue to present alarm notification on the local gateways.

A series of tests is performed with agents executing on $|G| = 12$ logical nodes. Operation is verified by monitoring agent messages displayed on the console. Node failure is simulated by terminating the agent gateway process. Termination of a gateway where agents reside also terminates those agents. The minimum number of nodes required for CONFIDANT failure is illustrated in Figure 23.

At time $t < 0$ all agents reside on unique gateways. Agents are dispatched at time $t = 0$. Here a single agent a_1^1 travels to a gateway hosting another agent a_2^1 as described previously in order to prevent deadlock. The first gateway terminated is the one hosting a_1^1 and a_2^1. At that point, the remaining $|G| - 1 = 11$ operational gateways are hosting $|A| = |G| - 2 = 10$ agents. As subsequent gateways fail, agents $a_{j \neq 1,2}^1$ are unable to access terminated members of committee C^1 and are unable to travel to a disabled gateway. Each case results in alarm notification from the remaining agents. During testing, the observed alert count decreases more rapidly than the theoretical alarm count. This is due to approximately one in three agents attempting to travel to a terminated gateway not accurately reporting an alarm upon the expiration of Δt_d.

Evaluation for Goal-2

In addition to eliminating a single point-of-failure, CONFIDANT is designed to mitigate the defined tampering modes using the techniques described previously. Eliminating the single point-of-failure is not only a concern for robust operation, but also plays a role in mitigating insider tampering. If a framework has a single point-of-failure, tampering at a single node could violate the integrity of the entire IDS.

Role of Manageability for Insider Tampering

Manageability is often used as a qualitative measure of IDS operation. While it is not the case that an IDS that is difficult to administer is robust against insider tampering, any IDS that is easy to manage is certainly not robust against insider tampering. Rather, for an IDS to minimize insider tampering exposures it is essential that it not be easily reconfigured. In fact, a centralized management console as found in CCSD and CCDD architectures represents a single point-of-failure as an insider could tamper with all monitored nodes from a single host. Consider the role of Tripwire Manager [21] as a centralized management console for Tripwire Servers. Tripwire Manager allows an administrator to distribute policy data and apply changes to servers distributed across the network.

The following test cases are designed to illustrate how an insider can make modifications, some management related, and defeat file integrity verification scans.

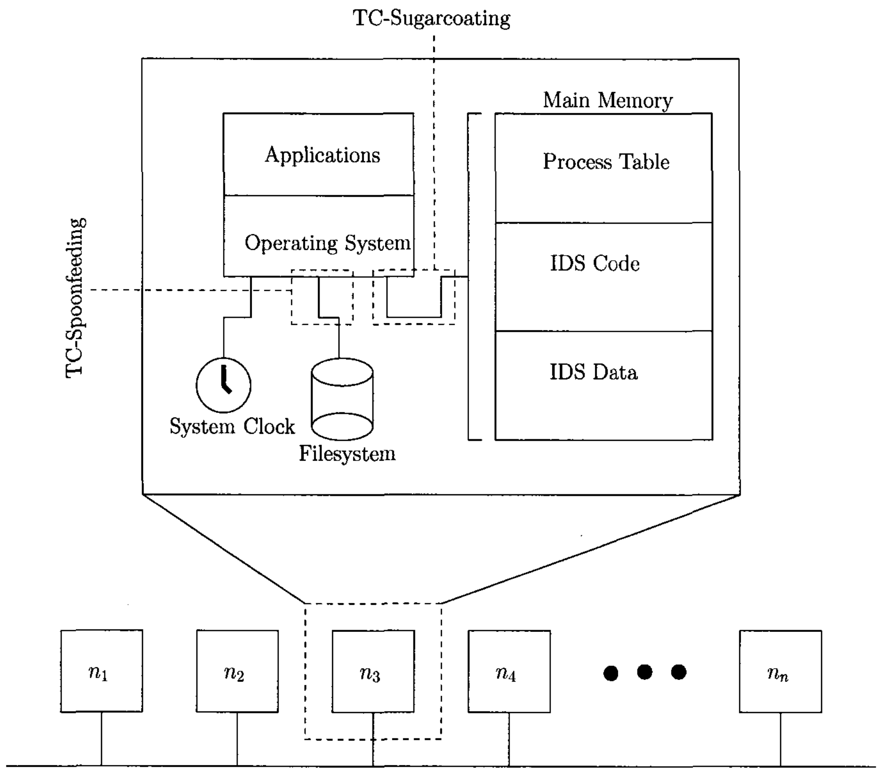

Figure 24: Spoonfeeding and Sugarcoating Tampering Points

Test cases are designed to illustrate how an insider could defeat the existing tools Tripwire and AIDE, compared to the CONFIDANT response to the same stimulus.

Testing of Spoofing Tampering Modes

Spoofing tampering modes involve the transmission of counterfeit data to IDS components. Spoonfeeding of data occurs at the sensor layer, Sugarcoating of data occurs at the control layer, and Recanting occurs at the alarm layer. Figure 24 illustrates Spoonfeeding and Sugarcoating tampering points. Spoonfeeding occurs at TP_{FS} while Sugarcoating occurs at TP_{IC}.

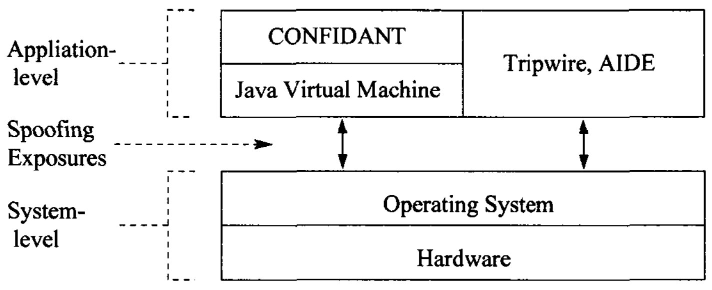

Figure 25: Spoofing Architecture Vulnerability

Test Case: TC-Spoonfeeding

Spoonfeeding data to file integrity sensor components involves the method by which file data is obtained. Existing tools obtain filesystem data via queries to the operating system. Using data provided by the OS allows tampering at the kernel or device driver level. Figure 25 illustrates injecting counterfeit data between the hardware and IDS levels.

Due to the separation between IDS functionality and system hardware, Figure 26 illustrates a Spoofing simulation. A random number generator is used to obtain values of 1 or 2. A value of 1 instructs the IDS to scan the valid data while 2 causes the IDS to scan the adversarial data stream. The test is performed ten times. Four times CONFIDANT scans the file and generates a TPI result. Six times it is redirected to the adversarial data and detects the expected MD5 resulting in a FNI. Spoonfeeding remains a vulnerability in the current version of CONFIDANT as filesystem data is obtained via operating system queries.

Accuracy of response is addressed by the assumptions listed in Table 6. A solution is inclusion of CONFIDANT file access routines in the kernel level and device driver level and is a topic of future research.

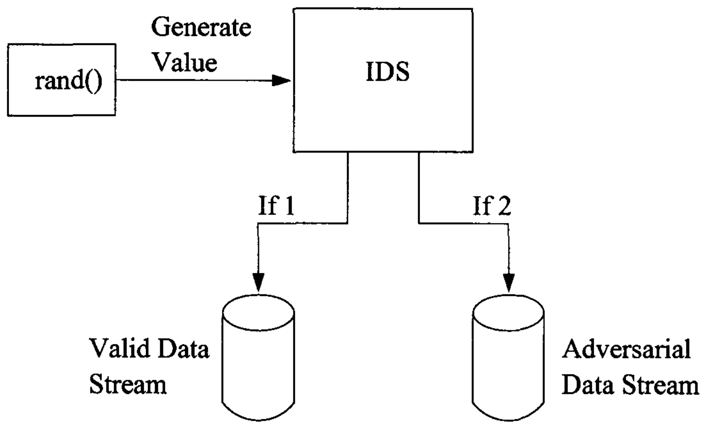

Figure 26: Spoofing Simulation Methodology

Test Case: TC-Sugarcoating

Most file integrity tools have the sensor and control layer closely coupled as to not be subject to tampering by sending false data to the control layer, referred to as Sugarcoating. Consider the case of Tripwire and AIDE. Sensor and control functions are performed by the same process. Tampering by the transmission of counterfeit data to the control layer would either have to be performed prior to the scan, which would be Spoonfeeding, or by modifications to the memory location used by the running process, which would fall under the alteration of internal data tampering modes. In the case where sensor and control layers illustrated in Figure 14 are comprised of physically distinct agents, Sugarcoating is mitigated in CONFIDANT by agent interlocking described previously.

Test Case: TC-Recanting

Recanting involves issuing counterfeit data to disable alert notification. This can exhibit a vulnerability if an alarm is recanted in a timely manner. Recanted alarms may not be fully investigated even if the alarm itself was legitimate. The tested tools do not have mechanisms to recant alarms. Once Tripwire and AIDE email messages arrive, there is no internal method to remove the alarm. It is possible for an insider to create an additional message stating that scan results are in fact valid, or remove the alarm message itself. Such techniques fall under the category of altering internal data as discussed below.

CONFIDANT mitigates tampering via Recanting by enforcing interlocking between agents. Alert messages are distributed to all committee agents when an error is encountered by a single member, so Recanting must be performed at every gateway simultaneously. Alarm notification arriving at one gateway, but not at others, is a sign of Recanting. Also, any interruption of communication between committee members results in an appropriate alarm. If an insider disables network connectivity in order to prevent alarm notification from reaching remote gateways, Δt_r will expire causing committee members a_j^i to report that an agent $a_{k \neq j}^i$ is unavailable.

Testing of Termination Tampering Modes

Tampering by termination involves the physical disabling of IDS components. Blindfolding, Commandeering, and Soundproofing are termination-based tampering modes.

Test Case: TC-Blindfolding

Blindfolding an IDS involves disabling sensor processes. Since the Tripwire and AIDE processes perform sensor and control routines, termination of the scanning process is tampering by Blindfolding. The `killall` utility is used to terminate IDS processes. This causes a race condition where if the verification process is able to complete operations before the script executes to disable operation, the scan will complete successfully. Knowledge of scan times t_{scan} increases the viability of Blindfolding as attempted process termination need not occur continuously. In order to facilitate testing, scans are performed every minute. Blindfolding of CONFIDANT is evaluated by termination of the gateway process.

The `cron` daemon emails any process execution output as a report to the task owner. AIDE generates a specific email alarm response while Tripwire relies on the `cron` generated output. Termination of the Tripwire process provides the following:

```
run - parts : /etc/cron.test/tripwire exited because of uncaught signal 9.
```

An email response is still generated when the Tripwire scan process is terminated, but simply displays an error message in place of the normal scan result. Normal AIDE operation results in an email alarm. Termination of the AIDE process using the `killall` utility does not allow the integrity scan to complete. Consequently, no email alarm message is generated. Even though there is no AIDE-specific response, `cron` reports:

```
run - parts : /etc/cron.test/aide exited because of uncaught signal 9.
```

In this case, the expected alarm is never observed, but insight into potential tampering is provided. One problem is that the response is generated by `cron` and not AIDE. It is possible that an administrator, by expecting an AIDE-specific response, will ignore the `cron`-generated notification. Since the Tripwire and AIDE processes are spawned by `cron`, another Blindfolding technique is to terminate the

Table 15: Termination Test Results

	Tripwire	**AIDE**	**CONFIDANT**
No Tampering	Cron Output (TPI)	AIDE Output (TPI)	Alarm (TPI)
Terminate IDS Process	Cron Error Output (TPI)	Cron Error Output (TPI)	Remote Alarm (TPI)
Terminate Mail Process	Cron Error Output (TPI)	Cron Error Output (TPI)	Alarm (TPI)
Terminate Cron Daemon	No Alarm (FNI)	No Alarm (FNI)	Alarm (TPI)
Sensitivity	0.725	0.725	1
Specificity	1	1	1

`cron` daemon. Termination of `cron` results in the absence of a response. This is expected because if the scan is never initiated, no response can be generated.

Termination of the CONFIDANT gateway process results in the expiration of Δt_d during an attempted agent dispatch. If a_j^i is unable to contact gateway g_m, alarm notification is sent to other members $a_{k \neq j}^i$ of committee C^i, where $1 \leq k \leq |C^i|$. Similarly, when file integrity scan results are unavailable, as is the case when agent a_j^i is terminated, other agents $a_{k \neq j}^i$ trigger an alarm upon expiration of Δt_r.

A random number generator was used to select between the absence of tampering, termination of the IDS, mail, or `cron` processes over many tests. Results are listed in Table 15. Of the three tested frameworks, CONFIDANT is most resistant to Blindfolding as an alarm is generated if a scan cannot be performed. In the best case, termination of Tripwire and AIDE processes will report that an error has occurred. In the worst case, termination of the `cron` daemon causes Tripwire and AIDE to fail completely. Termination of a CONFIDANT gateway prompts remote committee agents to report that a resource is unavailable as described previously.

Test Case: TC-Commandeering

Due to sensor and control functions being performed within a single process, the testing of Commandeering follows the same steps as Blindfolding, and has the same result listed in Table 15. Here again those processes are terminated prior to generation of alarms. Tripwire and AIDE either report error messages or have no response at all, while CONFIDANT generates an alarm as the scan cannot be performed.

Test Case: TC-Soundproofing

Soundproofing involves disabling of alarm components. Tripwire and AIDE utilize email for alarm purposes, while CONFIDANT uses communication and interlocking between agents to provide alert information via email. Using the `killall` utility to terminate email response results in a single notification for both Tripwire and AIDE. Table 15 lists the response in the presence of disabling email services. Since Tripwire utilizes `cron` for email alarms, termination of email services has no effect on response. The response from AIDE, however, does not exist in the same manner as described in TC-Blindfolding. The AIDE-specific response cannot be generated, and output is handled by `cron`. Termination of the mail process, even when the file under inspection is not modified, results in a message provided by `cron` stating that "aide has returned many errors." Termination of cron as described in TC-Blindfolding results in no response of any kind from either Tripwire or AIDE.

Since CONFIDANT does not rely on email for alarm notification, disabling of email services had no effect on alarm operation. Disabling of alarm components involves termination of a gateway on a specified host. Such behavior is recognized either by committee members executing on remote nodes or agents that attempt to travel to the terminated gateway as described previously.

Table 16: Blockading and Pacing Test Results

Tampering Mode	IDS		
	Tripwire	AIDE	CONFIDANT
Blockading	TP delayed up to 198s	TP delayed up to 193s	TP
Pacing	FN	FN	TP

Testing of Sidetracking Tampering Modes

A key aspect of tampering by Sidetracking is that the IDS components themselves are unmodified. Tampering by Blockading, Pacing, or Scapegoating do not modify the respective sensor, control, or alarm components. Each tampering mode was tested, and results for Blockading and Pacing are listed in Table 16. Details are described below.

Test Case: TC-Blockading

Tampering via Blockading involves isolating a sensor from needed access to a monitored file or device. Tampering point TP_{FS} was utilized for testing by physically removing the file. In this case, all tested frameworks detect that the file is unavailable with ideal sensitivity.

Another technique is to increase the system load using a high priority process at TP_{PT} in order to prevent the request for a filesystem scan from being serviced. Testing involves using the `stress` program [70]. Stress is a tool to impose load on a computer system including CPU, I/O, virtual memory, and disk stress. The maximum sustained load was approximately 45 as reported by the `uptime` system utility. Table 17 lists the system load and response time of a single test. In this test case, system load was gradually increased to the maximum sustained load. During this time, Tripwire and AIDE were scheduled to perform scans every minute while CONFIDANT agents traversed the network. Additional tests perform similarly.

Table 17: Blockading System Load and Alarm Delay

Load	Tripwire Delay (s)	AIDE Delay (s)
0.3	4	3
0.17	2	2
3.04	24	17
6.23	21	14
4.64	7	6
7.41	80	54
13.78	33	25
11.3	7	6
14.93	126	122
23.18	66	63
25.61	11	5
28.68	129	57
34.54	141	120
25.02	198	193
35.26	138	134
46.85	81	77
43.62	21	20
17.12	3	2
6.29	3	2

The results in Table 17 show some discrepancies between increased load and increased response time. For instance, upon reaching a load of 7.41, the Tripwire report is presented 80 seconds after the expected time. The following report is only delayed 25 seconds while under a load of 13.78. This is due to caching coupled with the short period between scans. Results do follow the general trend of increased delay under increased system load. The greatest delay encountered is 198 seconds while under a system load of 25.02.

While Tripwire and AIDE continued to generate reports under loads of 25-45, reports arrived out of order. Based upon execution order, AIDE reports are expected to appear first followed by Tripwire reports. Delays exceeding 60 seconds result in a group of AIDE reports followed by a group of Tripwire reports. Also, in approximately 30% of reports generated while the load was above 25, Tripwire is unable to complete the scan operation. CONFIDANT agents signal connection alarms as host services could not be accessed in a timely manner under any load imposed by `stress`.

Blockading of IDSs with network components can also be performed by increasing network load to forestall access at TP_N. The benefits of mobile agents described in previously include imposing minimal execution overhead, communication cost reduction, and a reduced network load compared to traditional client-server techniques. In order to investigate CONFIDANT network performance, agents are dispatched in the presence of an increasing network load as described below.

The previous discussion on network load illustrates that care must be taken when generating synthetic network traffic for testing network IDSs. Artificially generated traffic is not well-suited to test network intrusion detection accuracy, but rather is suited to test network hardware. Since the file integrity problem is host-based as it focuses on data residing on a local filesystem as opposed to traveling over the network, generating synthetic traffic to test throughput and latency is valid. The ability to operate in the presence of increasing network load as opposed to specific traffic is key.

In order to measure agent network performance, traffic is generated at a defined rate while agents traverse the network. Traffic is captured using `tcpdump` and written to a file. The traffic is replayed using `tcpreplay` with the rate parameter, `-r`, to specify the rate in megabits per second (Mbps) and the topspeed parameter, `-R`, to replay the traffic at the maximum rate. Agents obtain the current time on the local host, travel to a remote gateway, and return to the local host again obtaining the current system time. The difference between times obtained by the agent is the total round-trip dispatch and acknowledgment time for the agent to travel one hop in the network. Since agent network travel increases the overall

Table 18: Network Load and Agent Traversal Time

Load Specified by `tcpreplay` (Mbps)	**Load Observed by `tcpstat` (Mbps)**	**Average One-hop Round-trip Agent Traversal Time (ms)**	**Maximum Round-trip Traversal Time (ms)**
0	1.074652	221.33	0.383
1	1.677341	237.89	0.416
5	5.183018	215.55	0.249
10	9.410184	268.55	0.447
15	14.135512	294.33	0.491
20	18.255930	287.88	0.457
25	21.529032	590.00	3.463
Maximum	27.588796	658.11	3.241

network load, traffic rate is obtained using *tcpstat* reporting one second intervals. For each specified load, an agent performs repeated round-trip cycles, and the average traversal time was obtained as listed in Table 18.

As network traffic increases, the ability of agents to traverse the network is diminished. Network load up to approximately 20 Mbps results in a traversal time between 215 and 300 ms. The travel time increases significantly when load increases past 20 Mbps. Figure 27 illustrates the agent traversal time in the presence of increasing network load. Agent traversal times are relatively consistent up to a network load of 20 Mbps. High network loads greater than 20 Mbps steadily increase traversal delays. Under maximum network load of nearly 30 Mbps, agents are still able to traverse between gateways in under 700 ms. The interlocking nature of CONFIDANT agents requires that successful tampering occur at multiple gateways. Blockading increases agent dispatch delays and, thus, increases the time during which tampering can occur. For a network with n hops between the tampered node and the alarm destination, detection of a remote intrusion arrives within time δ by using an exponential spreading notification scheme where $\delta \geq (700ms)\log n$.

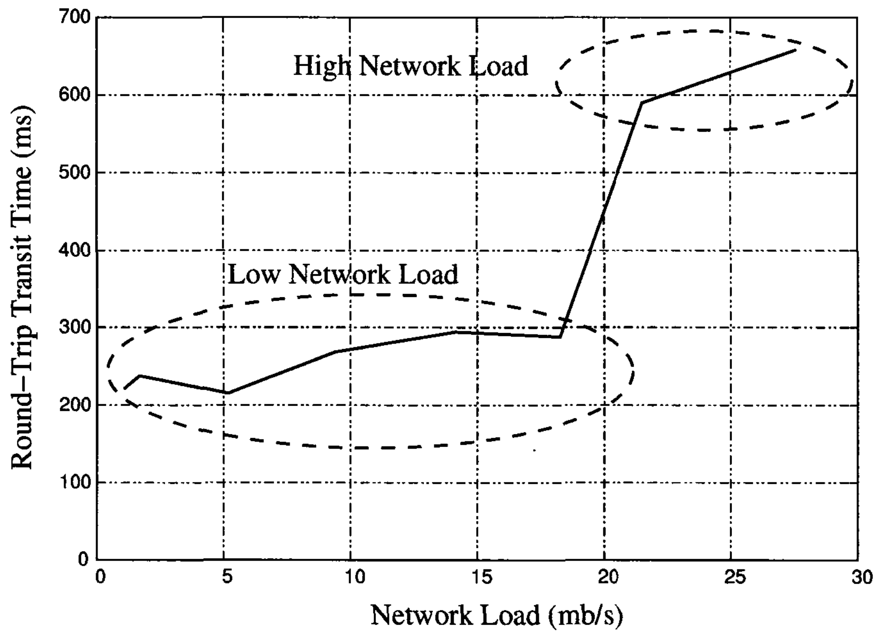

Figure 27: Agent Network Performance

CONFIDANT mitigates Blockading by taking network latency into consideration. Agent network performance testing provides accurate estimates of Δt values based on the physical network topology and expected load. A delay in excess of the appropriate Δt value is a potential sign of tampering as the remote gateway may be unreachable. Examples of Blockading network access include DoS attacks and physically disabling hardware resources. If an agent is unable to travel to the destination gateway prior to the expiration of the Δt_d time period, an alert is generated to inform security or administrative personnel of potential tampering, as shown in Figure 28.

Time values in Table 18 are an average value over a large number of iterations. The maximum traversal time of any single round-trip dispatch iteration under all tested network loads was 3.46 seconds. Based on this performance testing, it can be

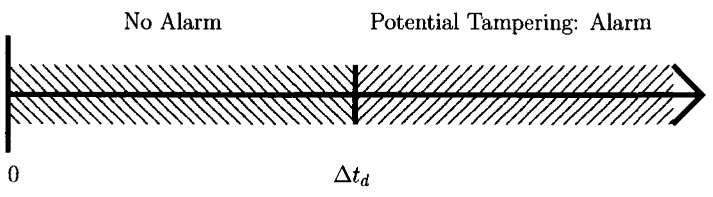

Figure 28: Blockading Timeline

seen that Δt_d values should be at least 3.5 seconds to take into consideration agent delays due to routine network traffic as shown in Figure 27. It is noteworthy that DCDD architectures are less prone to failure caused by network load issues due to the lack of dependence on any centralized resource as with other IDS architectures.

Test Case: TC-Pacing

Pacing involves tampering with external timing mechanisms. For tools that perform scans at times specified by the system clock, tampering can be performed by resetting the system clock. Figure 29 illustrates tampering by Pacing with a file integrity scan time of t_{scan} and time intervals τ. Initially the time-of-day clock and the actual time are the same, more formally $t = t_{TOD}$. When t enters the period $t_{scan} - \tau < t_{TOD} < t_{scan}$, the interval immediately preceding the file system scan, it is set to $t_{TOD} = t_{TOD} + 2 * \tau$ effectively skipping past the scan. Now the actual time appears to be in the interval $t_{scan} + \tau < t_{TOD} < t_{scan} + 2 * \tau$. When two τ periods have elapsed, t_{TOD} will be in the interval $t_{TOD} > t_{scan} + 3 * \tau$. The time-of-day clock is then reset to the actual time, $t_{TOD} = t_{TOD} - 2 * \tau = t$, file system scan has been bypassed, and the system clock has been restored to the actual time.

Tripwire and AIDE scans occur at periodic intervals based on the system clock and are consequently vulnerable to Pacing. During Pacing tests, Tripwire and

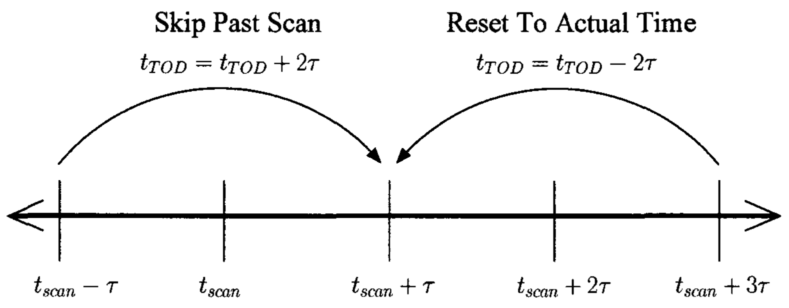

Figure 29: Pacing Timeline

AIDE were not able to make a single TPI detection. By employing internal timing delay mechanisms, CONFIDANT was able to detect all file modifications with perfect sensitivity and specificity and is not vulnerable to Pacing as listed in Table 16.

Test Case: TC-Scapegoating

Triggering alarms with the intent of overwhelming the alarm subsystem is Scapegoating. This essentially involves artificially increasing the false alarm count in order to divert the attention of security personnel away from the tampering. The significance of false alarms in intrusion detection is discussed previously. Scapegoating can be performed at TP_{FS} by writing multiple alarms to disk or at TP_{PT} by creating a process to generate additional alarms as listed in Table 19.

The main concern with Scapegoating is that alarm messages must be processed by the human administrator to verify their validity. Multiple alarms are generated in order to overload the administrator. An advantage that all file integrity tools have compared to network intrusion detection systems is the number of file scans, and consequently the low number of potential alarms should be relatively low. A network IDS may evaluate billions of packets each day. Based on the default operation, Tripwire and AIDE should produce a single report per day. The presence

Table 19: Scapegoating Technique Considerations

	Message-Centric	Process-Centric
Mode of Attack	Produce message data corresponding to additional alarms	Create a process to perform intrusive activity
Attacker Knowledge Required	Format of alarm messages	Existence of a vulnerability
Plausibility to Observer	Low to moderate	High
Tripwire Susceptibility	High	High
AIDE Susceptibility	High	High
CONFIDANT Susceptibility	Low	High

of hundreds of alarms, even without consideration for the content of the alarm, is a sign of an error at some level and indicates tampering via Scapegoating.

Tripwire and AIDE are subject to tampering via Scapegoating, as the result of file integrity scans are provided in email form. Simple text processing can be used to insert erroneous messages into a security administrator's inbox. Appending the existing mail file to itself, forming a new file twice as large as the original, can increase the number of false alarms exponentially. The current version of CONFIDANT employs messages displayed on the local console as well as messages transmitted to agents. It is possible to enable a process to present alarm messages on gateway consoles, but the complexity is increased due to the distributed and multi-agent nature of CONFIDANT. For instance, errors detected by a local agent will be relayed to remote agents within the committee, so the absence of corresponding alerts on remote nodes is an indication of Scapegoating. Also, multiple committee agents $a_1^1 \ldots a_n^1$ visit the node in question so alarms will be confirmed by multiple agents. Alarm messages from only one agent a_i^1 without corroboration from other agents in committee C^1 within the specified time window are indicative of Scapegoating.

Table 19 lists various technique considerations in message-centric and process-centric categories. In order to tamper by Scapegoating, an insider can either create a process to generate an alarm or create a process to perform intrusions resulting in alarms. Message-centric techniques only require knowledge of alarm format while process-centric tampering requires knowledge of an existing vulnerability. This helps process-centric techniques to appear to be more convincing than message-centric tampering. Due to the reliance on email for alarm notification, Tripwire and AIDE are particularly susceptible to message-centric techniques while CONFIDANT is not. Since alarm messages are distributed to remote nodes, successful Scapegoating in CONFIDANT requires significant additional effort to distribute appropriate erroneous alarm messages. Process-centric techniques, however, perform local intrusions and rely on agent interlocking to distribute alarm data to remote gateways. This increases convincingness as alarms are generated in response to an actual intrusion.

Testing of Altering Internal Data Tampering Modes

Tampering with internal data involves after-the-fact modification of an IDS component that is completely installed and properly configured prior to misuse. Unlike previous tampering modes, altering internal data tampering interacts directly with the IDS without termination of any of its logical components. Retroactive Baselining, Descoping, and Value Jamming are the associated tampering modes.

Test Case: TC-Retroactive Baselining

File integrity tools compute a baseline value for monitored files when the host is in a safe state for comparison with future integrity scans. Tripwire and AIDE store baseline values in a database file. Baseline data in CONFIDANT is internal to the mobile agents. Tampering by Retroactive Baselining involves modification

Table 20: Effort and Outcome Estimates for Tampering via Altering Internal Data

Tampering Mode	**IDS**		
	Tripwire	**AIDE**	**CONFIDANT**
Retroactive Baselining	Single operation < 5 minutes effort *10 of 10 resulted in FNI*	Single operation < 5 minutes effort *10 of 10 resulted in FNI*	Multiple operations > 8 hours effort *10 of 10 TPI detected by remote agents*
Descoping	Single operation < 5 minutes effort *10 of 10 resulted in FNI*	Single operation < 5 minutes effort *10 of 10 resulted in FNI*	Multiple operations > 8 hours effort *10 of 10 TPI detected by remote agents*
Value Jamming	Single operation < 5 minutes effort *10 of 10 resulted in FNI*	Single operation < 5 minutes effort *10 of 10 resulted in FNI*	Multiple operations > 8 hours effort *untested*

of the baseline value. Both Tripwire and AIDE provide mechanisms to reinitialize or update the local baseline database. The documentation for each describes an `--init` parameter to enable an insider to reinitialize the local database and an `--update` flag to allow for differences between existing database and current file state to be reconciled.

Tripwire baseline data is stored in the local database file `hostname.twd`. Executing the command *tripwire -init* will reinitialize the local database to reflect updated hash value of the modified files. Tripwire does require that a local passphrase, not the administrator password, be entered prior to updating the database facilitating mitigation of insider risk by differentiating an administrator from an observer. With updated baseline data, Tripwire is unable to recognize tampering with the monitored file. Tripwire documentation states that the baseline database should be stored on read-only media to mitigate tampering by outsiders. A read-only baseline database does not mitigate insider risk as an insider with physical access could replace or reconfigure the media. Updating the baseline database results in a False Negative Intrusion as modifications are not detected.

AIDE stores its local database as `aide.db`. Executing the command `aide --update` creates a new database, `aide.db.new`. Creating the new database, then copying it to the default location, is the method used by administrators to reflect approved file system changes. If AIDE is installed from a package in Debian Linux, the command `dpkg-reconfigure aide` will prompt the user to initialize the database and copy the new version to the default location. In both cases, Retroactive Baselining of Tripwire and AIDE is a system manageability issue. Enabling administrators to update the baseline database provides the defined method to facilitate insider tampering. As with Tripwire, once baseline data is modified in AIDE, no valid tampering detections were made and a False Negative Intrusion is encountered.

In order to tamper with CONFIDANT agents via Retroactive Baselining, the baseline data contained within the agent must be modified while in memory. Due to the dynamic nature of CONFIDANT agents, successful tampering requires that an attacker must:

- physically locate every agent within a committee across the network,
- determine the baseline memory location in each agent on the local and remote hosts, and
- update the memory location for all agents between message exchange and prior to dispatch.

Since the current version of CONFIDANT is written in Java, one technique that can be employed to modify memory contents is use of the Java debugger `jdb`. Compiling Java bytecodes with the `-g` flag and executed with the `-debug` flag enables the Java debugger to connect to the executing JVM using the password supplied on execution. Once the debugger is initialized, classes loaded in memory can be inspected. While the documentation states that process inspection can occur using jdb, tests were unsuccessful. Another technique involves writing to the contents of the `/proc` directory in Unix-based gateways. Tampering using these techniques was also unsuccessful.

In order to test the CONFIDANT response to Retroactive Baselining, internal modifications are simulated as illustrated in Figure 30. Two agents, a_1^1 and a_3^1, perform filesystem scans and post an event to agent a_2^1. The baseline value in a_1^1 is valid while the value in a_3^1 has been tampered with. The simulation involves the modification being present prior to initial dispatch as opposed to altered while executing. Both agents a_1^1 and a_3^1 send the message:

$$sendmsg.(a_2^1,\ MD5OK)$$

to agent a_2^1 stating that the scan result is negative. The internal baseline MD5 value is passed as part of the `MD5OK` event. Agent a_2^1 detects a discrepancy between its internal baseline and the one from agent a_3^1. An alarm is triggered and dispatched to all members of committee C^1. File modification is detected using propagation of alarm notification as previously illustrated in Figure 18.

If only a single agent a_3^1 is subjected to tampering, other agents within the committee C^1 will detect the baseline discrepancy and generate an alarm. An adversary must simultaneously determine the memory location of every agent within a committee on distributed nodes in the monitored network. Testing is simulated by having modified baseline data contained within one committee agent upon initial dispatch. While this agent considers a modified file to be valid, interlocking

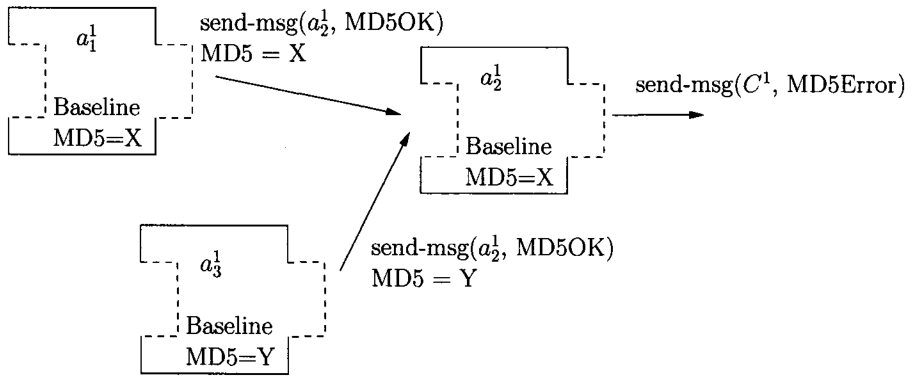

Figure 30: Agent Interaction in the Presence of Retroactive Baselining

messages between other committee members results in an alarm due to incongruent baseline data between committee members.

Test Case: TC-Descoping

Descoping differs from Retroactive Baselining in that integrity scan policy is modified as opposed to the scan baseline values. File integrity scan policy data specifies the files to be scanned. In order to modify the policy in Tripwire or AIDE, the associated configuration file must be edited, then the appropriate command executed. Tripwire includes the `--update-policy` option specifically to update the binary policy file from the text configuration as well as synchronize the baseline database with the updated policy data. In AIDE, the configuration file is modified to reflect updated policy information, then the `--update` command used to update the database.

In order to modify policy data in Tripwire, a policy database file must first be created. This is performed using the command `twadmin --create-polfile twpol.txt`, where `twpol.txt` is a text file containing policy configuration. The database must then be reinitialized using `tripwire --init`. Here the user must enter the site passphrase. As described in the TC-Retroactive Baselining test case for the local passphrase, the site passphrase may not be known, but it is easily modified by an insider. While removing an entry from the policy file will prevent the file in question from being scanned, the email notification does list the files that were scanned. A cognizant observer notices that a file to be monitored is missing from the scan report, but in the presence a large scan with many entries, such information may be overlooked.

Descoping in AIDE is performed by editing the configuration file, `aide.conf`, to reflect updated policy data, then reinitializing the database as described for the TC-Retroactive Baselining test case. Removing the appropriate entry from the configuration file and reinitializing the database prevents the previously monitored files from being scanned, and future file modifications are undetected.

As with the baseline data, all CONFIDANT agent policy information is stored internally. The same steps required to tamper by Retroactive Baselining are involved to tamper by Descoping with the exception of modifying policy as opposed to baseline data. Tripwire and AIDE specifically allow an administrator to reconfigure baseline and policy information, thereby allowing tampering by an insider. CONFIDANT has no such mechanism to update baseline or policy information.

Descoping testing is performed by simulation of modified policy data contained within one committee agent upon initial dispatch. The agent with modified policy data will omit scan of a monitored file. Remote agents that are expecting scan results will generate an alarm if results are unavailable.

In order to test the CONFIDANT response to Descoping, internal modifications are simulated as illustrated in Figure 31. Two agents, a_1^1 and a_3^1, perform filesystem scans and post an event to agent a_2^1. The baseline value in a_1^1 is valid while the

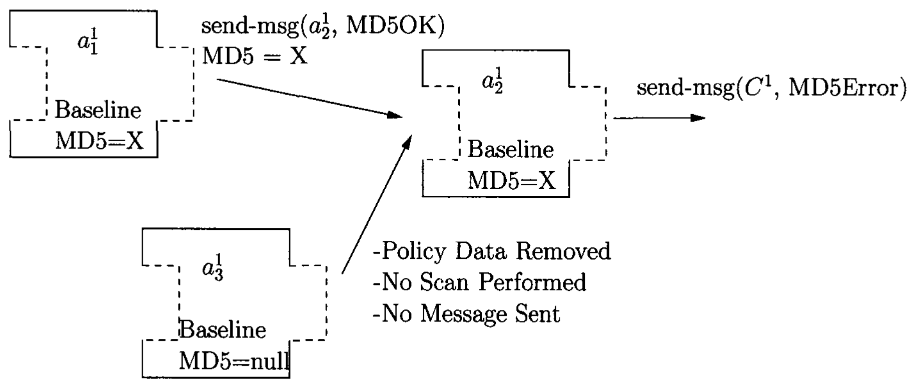

Figure 31: Agent Interaction in the Presence of Descoping

value in a_3^1 is `null`, as the policy data is modified to omit the scan. Agent a_1^1 sends the message:

$$sendmsg.(a_2^1,\ MD5OK)$$

to agent a_2^1 stating that the scan result is negative. Agent a_3^1, however, does not send a message as no scan is performed due to removal of policy data. Agent a_2^1 expects a scan status message from agent a_3^1. Since no message from a_3^1 is received by a_2^1 prior to time Δt_r expiring, agent a_2^1 sends the message:

$$sendmsg.(C^1,\ MD5Error)$$

to other members of committee C^1 to acknowledge that tampering has occurred. Alarm messages are propagated as previously illustrated in Figure 18.

Test Case: TC-Value Jamming

Value Jamming involves altering internal data in some way so that alarms are ignored. One technique is to write `FALSE` to a status location in memory as to indefinitely delay alarm notification. A less involved technique specifically for tools that employ email as the alarm mechanism is to modify or delete email contents. Once Tripwire and AIDE have delivered email messages detailing the result of the daily file integrity scan, the email can either be modified to reflect that file integrity is intact or it can be replaced with a copy of a previous message with updated header information. Successful tampering will effectively eliminate any alarms.

Value Jamming in CONFIDANT employed the same steps listed in TC-Retroactive Baselining to modify memory locations to disable agent messages. Messages in CONFIDANT serve as both communication and alarm notification. Disabling of alarm messages will also disable communication messages. In this case, the CONFIDANT response to Value Jamming is the same as the response for Descoping. In TC-Descoping, policy data is removed and a scan message is not received by agent a_2^1 prior to the expiration of Δt_r. When messaging is disabled, Δt_r will again expire prior to an expected message being received, thus activating the propagation of alarm notification.

Figure 32 illustrates Value Jamming in CONFIDANT by continuously asserting an internal memory modification so that scan messages always send a `MD5OK` event. This is transmitted with the expected MD5 value, even if the internal baseline data is invalid. Here agent a_3^1 contains modified baseline data. In the case of Retroactive Baselining, this was detected as the MD5 value in the message from a_3^1 does not match that of a_2^1, so alarms were generated. In this case, the MD5 value passed by a_3^1 to a_2^1 is modified to be the same as that in a_2^1 even though the internal baseline of a_3^1 is invalid.

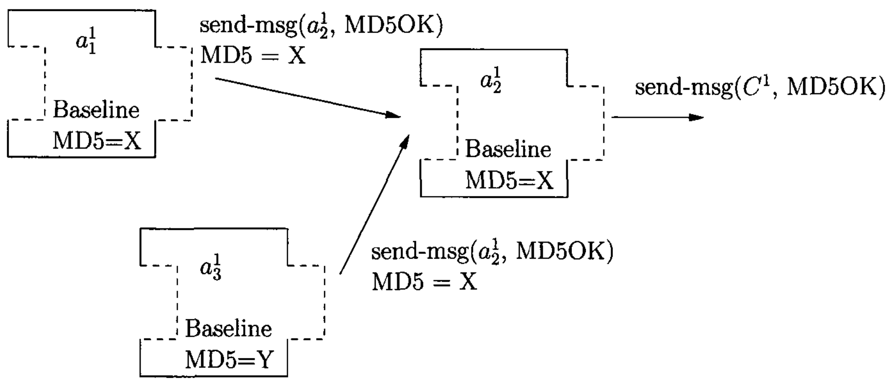

Figure 32: Agent Interaction in the Presence of Value Jamming

Successful Value Jamming in a single agent results in passing valid scan messages without regard for the results of the scan. This prevents alarm messages from being generated. Subsequent gateway visits by other agents in C^1 provided alarm notification as they remained unmodified. A successful adversary must simultaneously tamper with each agent in committee C^1. Testing of continued modification of agents in committee C^1 was unsuccessful. Thus, Value Jamming is mitigated in CONFIDANT by employing spatially and temporally distributed agents. Multiple agent visits on each gateway utilize a range of memory addresses. Also, multiple agents may reside on a gateway simultaneously. These efforts prevent tampering by modifying of a single memory location from being successful.

Testing of Selective Deception Tampering Modes

The ability to accurately predict integrity scan intervals is required to perform undetected tampering by File Juggling, as previously illustrated in Figure 5. Consider a scan time of t_{scan}, and a time period τ. Once t_{scan} is determined, operations can be performed before and after the scan in order to hide tampering. A file integrity scanner is susceptible to tampering by File Juggling if a pre-scan operation at time $t_{scan} - \tau$ and post-scan operation at time $t_{scan} + \tau$ can successfully hide file modifications.

Test Case: TC-File Juggling

File Juggling is performed by executing pre-scan operations to present filesystem data in the valid state in conjunction with post-scan attack operations. For the file integrity problem, these operations are copying valid data to the scan location prior to the scan, then replacing it with the modified data after the scan, as shown below. Tripwire and AIDE scan times are readily determined by inspection of the `cron` daemon configuration. File Juggling is illustrated in Figure 33. Scans are scheduled to occur at time t_{scan_1} and t_{scan_2}. The scan interval, t_{scan_2} - t_{scan_1}, in the default Tripwire and AIDE install is one day. The interval is decreased to five minutes for testing purposes. At time $t_{scan} - \tau$, the command:

```
cp confidant – valid.test confidant.test
```

is executed in order to provide the valid file to scan operations. At time $t_{scan} + \tau$, the command:

```
cp confidant – modified.test confidant.test
```

is executed replacing the valid file with the maliciously altered version. For testing purposes, scans are performed every five minutes with the interval τ set at one minute. Due to the use of periodic scan intervals coupled with copy operations performed before and after the scan, Tripwire and AIDE were unable to detect file tampering as the valid file was presented during scan operations. As described

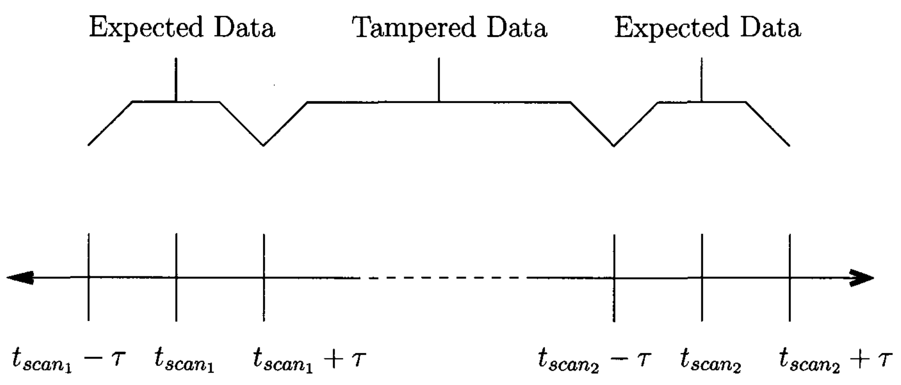

Figure 33: Expected and Tampered Data Presented During File Juggling

in the TC-Pacing test case, CONFIDANT does not rely on the system clock for timing information and scans are neither regularly scheduled or predictable. CONFIDANT was able to detect file modifications with perfect sensitivity. As stated previously, it is important that an agent can begin the scan operation and obtain filesystem data prior to an operating system context switch. If an attacker can monitor a process list, detect process initialization, and perform operations prior to the agents obtaining filesystem data, modified data can be replaced with valid data prior to MD5 hash computation. This is addressed by the assumptions listed in Table 6.

Valid data is observed by the IDS during the intervals $t_{scan} - \tau < t < t_{scan} + \tau$. During the interval $t_{scan_1} + \tau < t < t_{scan_2} - \tau$, filesystem data has been tampered with. The probability of the filesystem data being in the expected valid state is:

$$p_v = \frac{(t_{scan_1} + \tau) - (t_{scan_1} - \tau)}{t_{scan_2} - t_{scan_1}} = \frac{2\tau}{\Delta t_{scan}} \tag{7}$$

while the probability of filesystem data being modified is:

$$p_m = \frac{(t_{scan_2} - \tau) - (t_{scan_1} + \tau)}{t_{scan_2} - t_{scan_1}} = \frac{\Delta t_{scan} - 2\tau}{\Delta t_{scan}} = 1 - \frac{2\tau}{\Delta t_{scan}} = 1 - p_v. \quad (8)$$

Increasing τ decreases the probability of the data being in a modified state. When Δt_{scan} is set to 24 hours per NIST guidelines, $p_v = 0.0069$ even if the scan takes as long as 5 minutes.

TME Weighting Scheme and IDS Comparison

A metric weighting model called the Tampering Mode Exposure (TME) weighting scheme is developed based on the metric evaluation strategy described in [53]. In order to compare the frameworks numerically, categories and weights are defined and results computed using Equation 9 with j categories and $i = n$ metrics in each category j. Six categories, $j = 6$, are defined including one for each of the five tampering mode classes and a management category adapted from the previous metric discussion. The assigned weights and rationale for weight selection are listed in Table 21. Weights are given values of 1 to 4 based on the relative significance of each metric based on the methodology in [53]. Higher values indicate greater capability for management metrics and increased significance of successful tampering for tampering mode metric classes.

$$S = \sum_{j=1,6} \left[\sum_{i=1,n} (U_{ij} * W_{ij}) \right] \quad (9)$$

Unweighted scores are listed in Table 22. Scores are assigned a value of 1, 2, or 3 to signify detection failure, a modified result, and correct operation, respectively, for the tampering mode classes. For instance, testing of Selective Deception

Table 21: TME Metric Weighting Scheme

Category	Name	Weight	Rationale
Management	Monitoring	4	Security personnel may not remain in a single location
	Configurability	2	Ease of configuration enables insider tampering
	Scalability	2	The test network contains few nodes
Spoofing	Spoonfeeding	2	Attack requires intricate OS-level modification
	Sugarcoating	2	
	Recanting	1	Relies on human administrator response
Termination	Blindfolding	3	Trivial attack pathway for any insider
	Commandeering	3	
	Soundproofing	3	
Sidetracking	Blockading	1	Successful attempts delay accurate results
	Pacing	2	Modifying scan timing can prevent scan from occurring
	Scapegoating	1	Relies on human administrator response
Alter Internal Data	Retroactive Baselining	4	Baseline changes can make tampered data appear to be valid
	Descoping	4	Policy changes can make tampered data appear to be valid
	Value Jamming	4	Eliminating alarms gives a false sense of security
Selective Deception	File Juggling	3	Predictable scan timing facilitates future tampering

resulted in Tripwire and AIDE generating false negatives, so they are assigned a score of 1. CONFIDANT provided accurate alarm notification and is assigned a score of 3. Scores of the management category metrics are assigned based on the individual significance of each exposure.

Monitoring specifies the ability to receive alarm notification from multiple locations. The distributed nature of CONFIDANT provides alarms across the monitored network domain and is assigned a score of 3. AIDE has no network capability and is assigned a score of 1. The use of Tripwire Manager allows alarms to be received at a central console, thus providing greater monitoring ability than AIDE, but not fully distributed as in CONFIDANT.

Configurability as it relates to insider tampering is discussed in the previous section. Tripwire and AIDE utilize configuration files that can be modified by an administrator and are assigned high scores. A CONFIDANT design consideration is to disallow configuration to eliminate certain insider tampering exposures.

Using the weights in Table 21 and the scores in Table 22, a comparison of the frameworks can be performed. The weighted results are calculated using Equation 9 and listed in Table 23. CONFIDANT compares favorably under the TME weighted model where Tripwire and AIDE score comparably to each other. The scores and weights of the TME model, and the categories it uses, can be adapted to evaluate the performance of other IDSs in a similar manner.

Table 22: TME Unweighted Scores

Metric	Tripwire	AIDE	CONFIDANT	Rationale
Monitoring	2	1	3	Alarm notification on multiple nodes
Configurability	3	3	1	Ability to reconfigure once deployed
Scalability	2	1	3	Overlapping agents vs centralized control
Spoonfeeding	1	1	1	Architectural vulnerability in Figures 24 and 25
Sugarcoating	1	1	1	
Recanting	3	3	3	Administrator response
Blindfolding	2	2	3	Test result listed in Table 15
Commandeering	2	2	3	
Soundproofing	2	2	3	
Blockading	2	2	1	Test result listed in Table 16
Pacing	1	1	3	
Scapegoating	2	2	2	Administrator response
Retroactive Baselining	1	1	3	Test result listed in Table 20
Descoping	1	1	3	
Value Jamming	1	1	3	
File Juggling	1	1	3	Reliance on system clock

Table 23: Weighted Result

	Tripwire	AIDE	CONFIDANT	Maximum
Score	65	59	103	123

CONCLUSION

Summary of Results

Intrusion detection systems serve to identify security breaches capable of compromising computer system resource or service integrity. File integrity analyzers are a subset of host-based intrusion detection systems that verify computer filesystem data. Table 24 lists the contributions made to fields of IDS design and insider robustness resulting from this dissertation. First, a classification of tampering modes was identified based on user capability. Then, existing frameworks were inspected to identify vulnerabilities resulting in an architectural taxonomy of IDSs. In response to the identified vulnerabilities, a mobile agent framework consisting of agent behaviors interacting across multiple echelons was designed to mitigate the defined exposures. Finally, a comparative metric weighting scheme was designed to evaluate the relative performance of CONFIDANT to other frameworks.

Table 24: Contributions of Dissertation

Application Area	**Technical Challenge**	**Approach Taken**	**Results**
User Capability and Insider Risks	Wide range of vulnerabilities in existing IDS frameworks	Defined fundamental tampering points and 13 tampering modes	A generally-applicable classification of insider tampering including Spoofing and Termination
IDS Design	Understanding IDS relative capabilities	Defined architectural taxonomy of IDSs	CCSD, DCSD, CCDD, and DCDD categories
Mobile Agent Approach to ID	Existing frameworks exhibit a single-point-of-failure and are subject to tampering by insiders	Defined a framework of behaviors and agent interaction	Sensor, control, and response echelons were developed and evaluated
ID Metrics	Need to assess relative performance	Developed an adaptable weighted metric model	TME weighting scheme consisting of valued scores and weights

Tampering modes are identified as attacks undertaken to corrupt an intrusion detection framework. IDS tampering modes can be divided into five broad categories defined as *Spoofing*, *Termination*, *Sidetracking*, *Altering Internal Data*, and *Selective Deception.* These categories can be further identified as tampering directed specifically toward IDS sensor, control, and alarm categories. This research presents a discussion of tampering modes present in network-based file integrity analyzers and introduces *CONFIDANT*, the *Collaborative Object Notification Framework for Insider Defense using Autonomous Network Transactions.* Design of CONFIDANT is based on two goals:

Goal-1: *Reduce single point-of-failure exposures in existing IDS frameworks*, and

Goal-2: *Increase barriers against insider tampering pathways.*

These goals were evaluated by:

1. identifying single point-of-failure exposures in IDSs to address Goal-1,
2. developing a taxonomy of insider risks to address Goal-2,
3. designing metrics, weights, and experiments to quantify performance against both Goal-1 and Goal-2, and
4. comparing performance of the proposed and existing approaches using these metrics.

Testing was performed to illustrate the defined mitigation techniques. Tripwire and AIDE are evaluated in order to compare results with CONFIDANT's response. In the absence of tampering, all frameworks operate correctly. Results from tampering via Recanting, Scapegoating, and to some degree Value Jamming are similar among frameworks, as all rely on security administrator reaction to the presented alarm notification. Blockading causes all three frameworks to warn that resources are unavailable. Tripwire and AIDE reports arrive later than expected and out of order. Furthermore, termination-based tampering causes Tripwire and AIDE to fail completely, while CONFIDANT generates accurate alarm notification. CONFIDANT may be subject to tampering if an adversary is able to simultaneously modify all agents within a committee across a network domain. Attempts to perform such tasks have proven unsuccessful. Across all tampering modes, testing has shown that the CONFIDANT response is at least as accurate as the Tripwire and AIDE response to the same stimulus.

Testing has identified critical exposures in Tripwire, AIDE, and CONFIDANT as illustrated in Figure 34. Tripwire and AIDE exhibit critical exposures to tampering at TP_{SC}, TP_{ID}, and TP_{FS}. Specifically, they are highly subject to tampering via Pacing, all Altering Internal Data tampering modes, and File Juggling. Every test case for these tampering modes resulted in a FNI response. CONFIDANT exhibits a critical exposure at TP_{PT} but carries less significance than those for Tripwire and AIDE.

Testing of CONFIDANT shows that it is highly subject to Blockading. In fact, even under minimal system load, filesystem scans failed, and alarms were generated stating that the scan could not be performed. All three frameworks exhibited a critical exposure to Spoofing based on the evaluation discussion. The separation between the operating system layer and the application layer illustrated in Figure 25 allows an insider to tamper via Spoofing.

While each framework is subject to certain critical exposures, the severity of the associated tampering modes varies, as illustrated in Figure 35. The TME

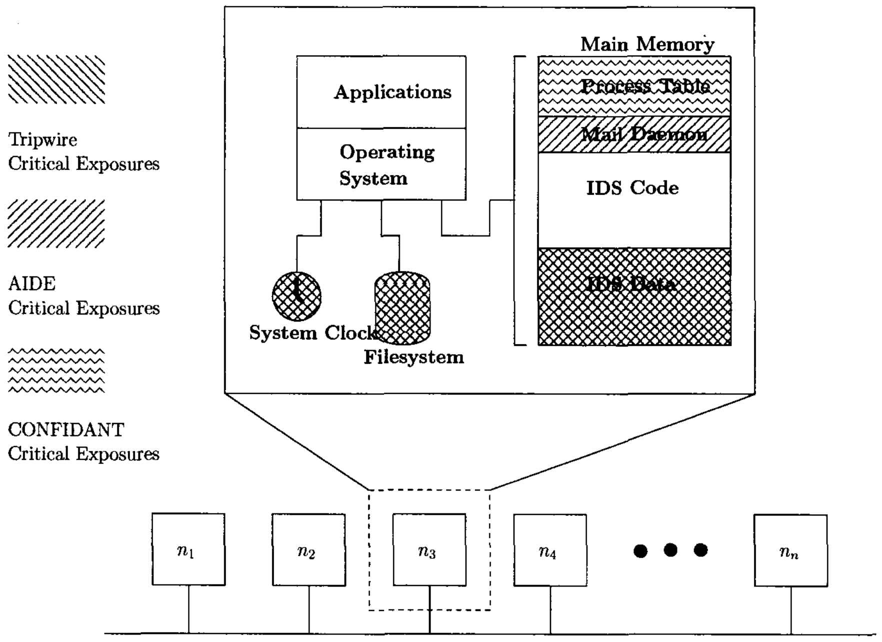

Figure 34: Tested IDS Critical Exposures

weights listed in Table 21 are based on tampering mode severity. For instance, tampering via Blockading is not as severe as tampering via Pacing. Blockading causes results to be delayed while Pacing has the potential to completely bypass scan operations. Similarly, Pacing is not as severe as Retroactive Baselining as updating the baseline database causes the IDS to interpret all results as valid, while configuration and scan timing remains unchanged and therefore undetected. Testing has shown that the Altering Internal Data tampering modes can be the most severe, while Scapegoating, Blockading, and Recanting are not as detrimental nor effective.

Testing also showed the relationship between configurability and robustness against insider tampering as illustrated in Figure 36. IDSs that provide configuration and management routines inherently enable insider tampering. This can best

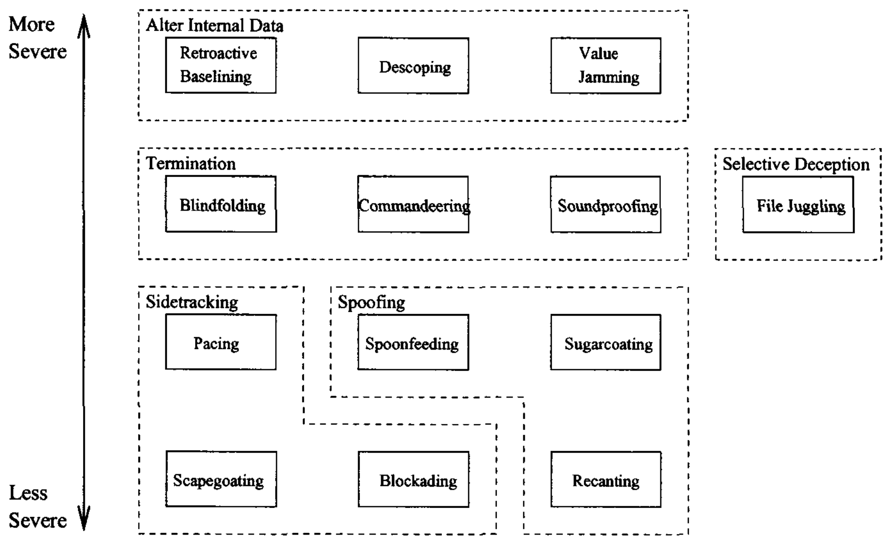

Figure 35: Relative Tampering Mode Impact

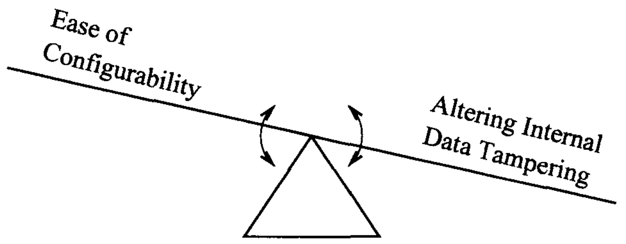

Figure 36: Configurability vs. Robustness Tradeoffs

be seen by inspection of the test results for the Altering Internal Data tampering modes. Consider Retroactive Baselining in Tripwire as opposed to CONFIDANT. Tripwire includes commands to allow an administrator to update baseline data. Once the data has been updated with a MD5 of a modified file, subsequent scans report that the file is valid. CONFIDANT does not have built in management routines, and thus an administrator can not easily update the internal baseline data. Testing has shown that tampering with one agent a^i_j in committee C^i results in other agents $a^i_{k\neq j}$ detecting that tampering has occurred.

CONFIDANT testing has shown it to be effective in mitigating several severe insider tampering exposures at the expense of manageability. A distributed design is essential for robust operation in the presence of insider tampering as it enforces successful tampering to occur at multiple nodes simultaneously. Two major difficulties with this design include the interlocking of distributed components and recovery upon intrusion detection. Agent interlocking is required to ensure that components remain distributed. Also, since manageability is sacrificed in order to enhance robustness against insider tampering, recovery after alarm notification required that CONFIDANT be restarted on all nodes. This may not be practical for large enterprise installations.

Cascading Tampering Modes

The previous discussion of user capability and the CONFIDANT test cases focus on individual tampering modes. Future work includes investigation of cascading tampering modes. Certain tampering modes may be combined to increase IDS tampering exposures. For instance, the susceptibility of an IDS to Selective Deception may be increased with resource blockades and high priority processes. Tampering by Pacing or Blockading may allow File Juggling to occur as illustrated in Figure 37. Successful File Juggling depends on the predictability of scan timing. An attacker could first perform Blockading to delay IDS access to filesystem resources and then preform File Juggling resulting in a successful attack. Similarly, Pacing can be performed to corrupt the system time by setting the system clock to a time when a scan is known to not occur in order to facilitate File Juggling. Another example involves tampering via Blockading in order to localize agents to

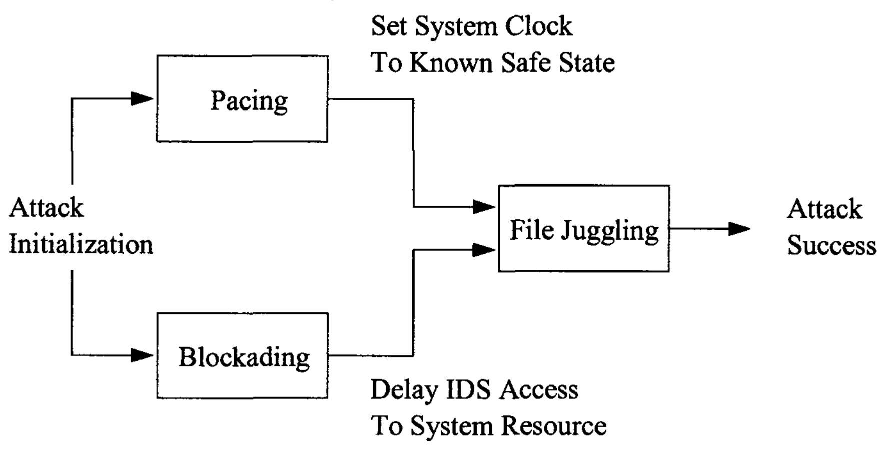

Figure 37: Cascading Tampering Mode Pathway Example

a smaller network domain than defined upon initial dispatch. This may increase the exposure to Termination tampering modes.

Future Work

CONFIDANT design and testing assumptions are based on the security of the agents. Security of mobile agents is a topic of current research [65] [63] [64]. It is critical that:

- deciphering of encrypted communication remains secure at channel endpoints,
- an agent is able to verify that the gateway being visited has not been tampered with,
- agents execute at a sufficiently low level to provide direct filesystem access, and

- filesystem access is obtained immediately upon agent arrival so processes are unable to detect that a scan is imminent.

Currently, CONFIDANT agents make requests to the operating system for file system data. This creates an avenue for tampering via Spoonfeeding as a malicious insider could enable kernel or driver functionality to provide false file data during scans. Possible techniques to solve this problem include compiling the gateway into the kernel or enabling the agents to perform integrity analysis on the gateway upon visitation.

Another remaining tampering exposure is that communication must be decrypted at endpoints in order to be processed. This provides an insider the ability to tamper with data upon arrival. For instance, tampering via Sugarcoating may occur between processes or agents on remote nodes. Communication between nodes or agents is handled by using SSL [71] for authentication and encryption as well as to prevent man-in-the-middle attacks [72]. Secure communication is ensured under SSL by strong cryptography. A vulnerability remains, however, in that encrypted data must be decrypted prior to use. Once it is decrypted, it becomes a potential avenue for tampering and is a future topic of research in secure microprocessor hardware [73].

In the current version of CONFIDANT, each agent maintains a list of available gateways including the location of each committee member. Prior to agent dispatch, a random number is generated to select a destination gateway. If the selected gateway is occupied, the agent will travel to the next available gateway. Future research includes modeling the network as a graph or Markov chain and performing statistical coverage testing to determine optimal agent transitions. A final topic of future research is to extend CONFIDANT capabilities to perform host-based IDS functions in addition to file integrity.

APPENDIX A
SAMPLE CONFIDANT AGENT ROUTINES IN JAVA

```
import java.io.IOException;
import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.util.Enumeration;
import java.util.Random;

import COM.meitca.concordia.Agent;
import COM.meitca.concordia.AgentTransportException;
import COM.meitca.concordia.Destination;
import COM.meitca.concordia.Itinerary;
import COM.meitca.concordia.LaunchException;
import COM.meitca.concordia.ServerUnavailableException;
import COM.meitca.concordia.event.EventHandlerException;
import COM.meitca.concordia.event.EventManagerConnection;
import COM.meitca.concordia.event.EventType;

public class ConfidantAgent extends Agent  {
     //this file is an illustrative example adapted from
     //the testing examples
     //NOTE: the code must be adapted to the physical
     //network topology to run properly
     //Also, it has been formatted to fit in this appendix

     //the event dispatched within a probe agent for robust
     //communication
     private ConfidantEvent probeevent = new ConfidantEvent();
     private String name = null;
     private String eventText = null;
     //internal baseline
     private String expectedMD5 = "50a1f6525d4ae14ecabfdb2fe8f03e0f";
     private String computedMD5 = null;
     //internal policy
     private String fileName = "confidant.test";
     private String destGateway = null;
     // the agent carries related classes during travel
     private String[] related = {"MD5Check", "ConfidantEvent"};

     private int gnum = 4; //default -- reset by method
     //glist is the list of monitored gateways
     //Concordia can't use a multi-dimension array or an array
```

```
//of objects with 2 string elements so 2 arrays are used
private String[] glistname = new String[gnum];
private String[] glistagent = new String[gnum];

private int anum = 2; //default -- reset by method
//list of committee agents and the gateways on which
// they reside
private String[] calistname = new String[anum];
private String[] calistgateway = new String[anum];

//response arrays to handle messages asynchronously
//as messages arrive, they are placed into these arrays
private String[] arrivalresponsemessage = new String[anum];
private String[] scanresponsemessage = new String[anum];
private String[] dispatchresponsemessage = new String[anum];

//delta times in milliseconds
private long delta_ts = 4000;
private long delta_td = 2000;
private long delta_tr = 8000;

//parameters are handled in the config file parser
//and agent launcher
public ConfidantAgent() {
     this.setRelatedClasses(related);
}

public void setName(String n) {
     name = n;
}

public String getName() {
     return name;
}

// This is called by the Concordia Server immediately
// prior to the agent being transported to its next
// destination. Its purpose is to perform any cleanup
// required before the Agent migrates to its next destination.
public void prepareForTransport() throws
            AgentTransportException {
```

```
          System.out.println("ConfidantAgent " + getName() +
                " prepareForTransport");
          // code that should go here is in completedTransport
// due to the reasons described below
   }

   // This is called by the Concordia Server when the
   // Agent arrives at its new destination. Its purpose
   // is to perform any initialization required by the
   // Agent when it arrives at a new destination.
   public void completedTransport() throws
               AgentTransportException {
          System.out.println("ConfidantAgent " + getName() +
                " completedTransport");

          makeEventConnections();

          System.out.println("Agent " + getName() + "Arrived");
          for (int i=0; i<gnum; i++) {
               if (glistname[i].equals(destGateway)) {
                    glistagent[i] = getName();
                    break;
               }
          }
          System.out.println("Sending AgentArrived Event");
          ConfidantEvent travelComplete = new ConfidantEvent();
          travelComplete.setType("AgentArrived");
          for (int i=0; i<anum; i++) {
               travelComplete.setFromAgent(this.getName());
               travelComplete.setToAgent(calistname[i]);
               travelComplete.setNote(destGateway);
               postConfidantEvent(travelComplete, calistgateway[i]);
          }

          delay(delta_ts);
          System.out.println("Processing Arrival Response");
          processArrivalResponse();

          delay(10000);  //slow things down to see what is going on

          //the following really should go in prepareForTransport
```

```
// but since Concordia agents hang trying to travel to a
// disabled gateway without posting an event (exceeding
// delta_td) dispatch must be handled manually

            System.out.println("Sending AgentTravelRequest Event");
            ConfidantEvent travelRequest = new ConfidantEvent();
            travelRequest.setType("AgentTravelRequest");
            for (int i=0; i<anum; i++) {
                travelRequest.setFromAgent(this.getName());
                travelRequest.setToAgent(calistname[i]);
                travelRequest.setNote(null);
                postConfidantEvent(travelRequest, calistgateway[i]);
            }

            delay(delta_ts);
            System.out.println("Processing Dispatch Response");
            processDispatchResponse();

            dispatch();
        }

        public void dispatch() {

            selectGateway();

            ConfidantAgent agent = new ConfidantAgent();
            //keep the same name as the creating agent
            agent.setName(this.getName());
            Itinerary itinerary = new Itinerary();
            itinerary.clearItinerary();
            Destination destination = new Destination
              ("rmi://localhost/" + destGateway, "scan");
            itinerary.addDestination(destination);
            agent.setItinerary(itinerary);
            agent.setGatewayList(glistname, glistagent);
            System.out.println("Launching agent " +
                agent.getName() + " to " + destGateway);
            agent.launch();
        }

        // This is called by the Concordia Server when the
// Agent completes its Itinerary. Its purpose is to
// perform any cleanup required after the Agent
// completes its itinerary.
public void completedItinerary() throws
             AgentTransportException{
     System.out.println("ConfidantAgent " + getName() +
           " completedItinerary");
}

public void launch() {
     System.out.println("ConfidantAgent " + getName()
         + " launching");
     try {
          super.launch();
     } catch (LaunchException e) {
          System.out.println("ALARM ------ Error dispatching
               agent " + getName());
          //if there is an error, print out message then
  //call prepareForTransport again
          dispatch();
     }
}

public void makeEventConnections() {
     System.out.println("ConfidantAgent
          makeEventHandler");
     this.makeEventHandler();
     System.out.println("ConfidantAgent
          makeEventManagerConnection");
     this.makeEventManagerConnection("localhost");
     System.out.println("ConfidantAgent
           registerAllEvents");
     this.registerAllEvents();
}

public void makeEventHandler() {
     //true is sync, false is async... creates a new thread
     try {
          super.makeEventHandler(false);
     } catch (EventHandlerException e) {
          System.out.println("Error: makeEventHandler "
                + getName());
            e.printStackTrace();
        }
    }

    public void makeEventManagerConnection(String host) {
        String url = EventManagerConnection.EventManager
           URL(host);
        try {
            //true uses a proxy, false makes a direct
      //connection
            super.makeEventManagerConnection(url, false);
        } catch (NoSuchObjectException e) {
            System.out.println("Error: makeEventManager
                Connection " + getName());
            System.out.println("NoSuchObjectException");
            e.printStackTrace();
        } catch (RemoteException e) {
            System.out.println("Error: makeEventManager
                Connection  " + getName());
            System.out.println("RemoteException");
            e.printStackTrace();
        }
    }

    // Register interest in receiving all events
    // Documentation says we can only register specific
    // events but this doesn't seem to work.
    public void registerAllEvents() {
        try {
            super.registerAllEvents();
        } catch (RemoteException e) {
            System.out.println("Error: registerAllEvents "
                + getName());
            System.out.println("RemoteException");
            e.printStackTrace();
        } catch (SecurityException e) {
            System.out.println("Error: registerAllEvents "
                + getName());
            System.out.println("SecurityException");
            e.printStackTrace();
    } catch (IOException e) {
        System.out.println("Error: registerAllEvents "
            + getName());
        System.out.println("IOException");
        e.printStackTrace();
    } catch (ServerUnavailableException e) {
        System.out.println("Error: registerAllEvents "
            + getName());
        System.out.println("ServerUnavailableException");
        e.printStackTrace();
    }
}

public void scan() {
    ConfidantEvent scanResult = new ConfidantEvent();
    computeStuff(); //stores result to computedMD5
    scanResult.setFromAgent(this.getName());
    scanResult.setNote(computedMD5);
    if (computedMD5.equals(expectedMD5)) {
        scanResult.setType("MD5OK");
    } else {
        scanResult.setType("MD5Error");
    }
    for (int i=0; i<anum; i++) {
        scanResult.setFromAgent(this.getName());
        scanResult.setToAgent(calistname[i]);
        scanResult.setNote(null);
        postConfidantEvent(scanResult, calistgateway[i]);
    }

    delay(delta_ts);
    processScanResponse();
}

public void processScanResponse() {
    for (int i = 0; i < anum; i++) {
        //null if there isn't a scan response
        if (scanresponsemessage[i] == null) {
            System.out.println("ALARM ---- Missing ACK
          from Scan Result");
            break;
        }
    }
    resetScanResponse();
}

public void processDispatchResponse() {
    for (int i = 0; i < anum; i++) {
        //null if there isn't a dispatch response
        if (dispatchresponsemessage[i] == null) {
            System.out.println("ALARM ---- Missing
         Response to Dispatch Request");
            break;
        }
    }
    resetDispatchResponse();
}

public void processArrivalResponse() {
    for (int i = 0; i < anum; i++) {
        //null if there isn't an arrival response
        if (arrivalresponsemessage[i] == null) {
            System.out.println("ALARM ---- Missing
         Response to Arrival Notification");
            break;
        }
    }
    resetArrivalResponse();
}

public void selectGateway() {
    Random rand = new Random();

    int ng = Math.abs(rand.nextInt()%gnum);

    while ( glistagent[ng] != null) {
        ng = Math.abs(rand.nextInt()%gnum);
    }
    destGateway = glistname[ng];
}

public void handleEvent(EventType event) {
      if (event instanceof ConfidantEvent) {
           handleConfidantEvent((ConfidantEvent)event);
      }
      else {
           handleUnknownEvent(event);
      }
 }

 public void handleConfidantEvent(ConfidantEvent event) {
      System.out.println("Handling ConfidantEvent");
      String responsetype = event.getType();
      if (responsetype.equals("AgentArrived")) {
           System.out.println("Handling AgentArrived
                Event");
           //send ACK
           ConfidantEvent ack = new ConfidantEvent();
           ack.setType("AgentArrivedACK");
           ack.setFromAgent(this.getName());
           //return to sender
           ack.setToAgent(event.getFromAgent());
           ack.setNote(destGateway);
           postConfidantEvent(ack, event.getFromAgent());
      } else if (responsetype.equals("AgentArrivedACK")) {
           System.out.println("Handling AgentArrivedACK
                Event");
           //store in array for processing to make sure
//messages are received for all committee
// agents
           for (int i=0; i<anum; i++) {
                if (calistname[i].equals
           (event.getFromAgent())) {
                     calistgateway[i] = event.getNote();
                     break;
                }
           }
      } else if (responsetype.equals("AgentTravelRequest")) {
           System.out.println("Handling AgentTravelRequest
                Event");
           //send Proceed, wait for notification on new gateway
           ConfidantEvent proceed = new ConfidantEvent();
           proceed.setType("AgentTravelProceed");
```

```
            proceed.setFromAgent(this.getName());
            proceed.setToAgent(event.getFromAgent());
            proceed.setNote(destGateway);
            postConfidantEvent(proceed, event.getFromAgent());
        } else if (responsetype.equals("AgentTravelProceed")) {
            System.out.println("Handling AgentTravelProceed
                  Event");
            //store in array for processing to make sure
//dispatch is confirmed by all committee
//agents
            for (int i=0; i<anum; i++) {
                 if (calistname[i].equals
            (event.getFromAgent())) {
                      dispatchresponsemessage[i] =
    event.getNote();
                      break;
                 }
            }
        } else if (responsetype.equals("MD5OK")) {
            System.out.println("Handling MD5OK Event");
            //does this result match our baseline?
            if ((event.getNote()).equals(expectedMD5)) {
                 System.out.println("MD5OK Event and matching
              baseline");
            } else {
                 System.out.println("ALARM ------ MD5OK Event
               but different baseline");
            }
        } else if (responsetype.equals("MD5Error")) {
            System.out.println("Handling MD5Error Event");
            //alarm
            System.out.println("ALARM ------ MD5 Error detected
                  by "+ event.getFromAgent());
        } else if (responsetype.equals("AgentUnavailable")) {
            System.out.println("Handling AgentUnavailable
                 Event");
            //can't contact agent -- alarm
            System.out.println("ALARM ------ Agent " +
                event.getToAgent() + " is missing");
        } else if (responsetype.equals("HostUnavailable")) {
            System.out.println("Handling HostUnavailable
                    Event");
                //can't contact gateway -- alarm
                System.out.println("ALARM ------ Gateway"
                     + destGateway + " is unavailable");
           } else {
                System.out.println("Handling ConfidantEvent of
                     Unknown Type"); //shouldn't ever get here
           }
}

      //shouldn't ever get to this
      public void handleUnknownEvent(EventType event) {
           System.out.println("Handling Unknown Event");
           System.out.println("Event ID:           " +
                 event.getEventID());
           System.out.println("Event Description: " +
                 event.getEventDescription());
           System.out.println("Event toString:     " +
                 event.toString());
      }

      public void postConfidantEvent(ConfidantEvent event,
                      String destination) {
           //probe agent behavior to do the posting
           ProbeAgent ag = new ProbeAgent();
           ag.setName("Probe");
           Itinerary it = new Itinerary();
           it.clearItinerary();
           Destination de = new Destination(destination,
             "probePost");
           it.addDestination(de);
           ag.setItinerary(it);
           ag.setProbeEvent(event);
           ag.setSendingAgent(this.getName());
           ag.launch();
      }

      public String computeMD5(String filename) {
           MD5Check mdc = new MD5Check();
           return mdc.computeDigestAsString(filename);
      }
```

```
public void computeStuff() {
     System.out.println("ConfidantAgent " + getName()
          + " is computing stuff");
     String str = new String(System.getProperty("os.name",
         "OS Name not found"));
     if (str.startsWith("Linux")) {
          String fn = new String("/home/" + fileName);
          System.out.println("Computing MD5 of " + fn);
          computedMD5 = computeMD5(fn);
          System.out.println("MD5 of " + fn + " is "
               + computedMD5);
          if (computedMD5.equals(expectedMD5)) {
               System.out.println("Hey, they match");
          } else {
               System.out.println("MD5 signatures don't match...
            possible tampering");
          }
     }
     else if (str.startsWith("Windows")) {
          String fn = new String("c:\\" + fileName);
          System.out.println("computing MD5 of " + fn);
          computedMD5 = computeMD5(fn);
          System.out.println("MD5 of " + fn + " is "
              + computedMD5);
          if (computedMD5.equals(expectedMD5)) {
               System.out.println("Hey, they match");
          } else {
               System.out.println("MD5 signatures don't match...
            possible tampering");
          }
     }
     else {
          System.out.println("I don't recognize the OS...
                now what do I do?");
     }
     System.out.println("ConfidantAgent " + getName()
              + " is done computing stuff");
}

public void delay(long time) {
    //time is milliseconds
    try {
         Thread.sleep(time);
    } catch (InterruptedException e) {
         System.out.println("Error: agent couldn't sleep
              for" + time + "ms");
         e.printStackTrace();
    }
}

public String printData() { //toString-like functionality
    StringBuffer sb = new StringBuffer();
    sb.append("Name: " + name + "\n");
    sb.append("ID: " + toString() + "\n");
    sb.append("Itinerary:\n");
    sb.append("Host: ");
    for(Enumeration e = getItinerary().destinations();
          e.hasMoreElements(); ) {
         Destination d = new Destination();
         d = (Destination)e.nextElement();
         sb.append(d.getDestinationHost() + "\t");
    }
    sb.append("\n");
    sb.append("Function: ");
    for(Enumeration e = getItinerary().destinations();
          e.hasMoreElements(); ) {
         Destination d = new Destination();
         d = (Destination)e.nextElement();
         sb.append(d.getMethodName() + "\t");
    }
    sb.append("\n");
    return sb.toString();
}

public void setGatewayList(String[] gn, String[] ga) {
    glistname = gn;
    glistagent = ga;
}

public void printGatewayList() {
    for (int i=0; i< gnum; i++) {
          System.out.println("Gateway name: " + glistname[i]
               + "  Agents: " + glistagent[i]);
     }
}

public void setAgentList(String[] an, String[] ag) {
          calistname = an;
          calistgateway = ag;
}


public void setArrivalResponse(String[] ar) {
          arrivalresponsemessage = ar;
}

public void resetArrivalResponse() {
          for (int i = 0; i < anum; i++) {
               arrivalresponsemessage[i] = null;
          }
}

public void setScanResponse(String[] srm) {
          scanresponsemessage = srm;
}

public void resetScanResponse() {
          for (int i = 0; i < anum; i++) {
               scanresponsemessage[i] = null;
          }
}

public void setDispatchResponse(String[] drm) {
          dispatchresponsemessage = drm;
}

public void resetDispatchResponse() {
          for (int i = 0; i < anum; i++) {
               dispatchresponsemessage[i] = null;
          }
}
```

```
public void setNumberOfAgents(int a) {
     anum = a;
}

public int getNumberOfAgents() {
     return anum;
}

public void setNumberOfGateways(int g) {
     gnum = g;
}

public int getNumberOfGateways() {
     return gnum;
}

public void setProbeEvent(ConfidantEvent event){
     probeevent = event;
}

public ConfidantEvent getProbeEvent() {
     return probeevent;
}

public void busywait() {
     System.out.println("Agent " + getName()
          + " performing a busywait for debugging
       purposes");
     while (true) {
     }
}

public void printMsg() {
     System.out.println("----------> printMsg <----------");
}

public static void main(String args[]) {
     ConfidantAgent agent = new ConfidantAgent();
     agent.setName("ConfidantAgent test agent");
     Itinerary i = new Itinerary();
     i.addDestination(new Destination("localhost",
            "printData"));
           agent.setItinerary(i);
           agent.launch();
           System.exit(0);
      }
}
```

```
import java.io.IOException;
import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import COM.meitca.concordia.Agent;
import COM.meitca.concordia.AgentTransportException;
import COM.meitca.concordia.LaunchException;
import COM.meitca.concordia.ServerUnavailableException;
import COM.meitca.concordia.event.EventException;
import COM.meitca.concordia.event.EventHandlerException;
import COM.meitca.concordia.event.EventManagerConnection;

public class ProbeAgent extends Agent  {

     private ConfidantEvent probeevent = new ConfidantEvent();
     private String name = null;
     private String destGateway = null;
     private String destHost = null;
     private String[] related = {"MD5Check", "ConfidantEvent"};
     private String sendingagent = null;
     private String sendinggateway = null;

     public ProbeAgent() {
          this.setRelatedClasses(related);
     }

     public void setName(String n) {
          name = n;
     }

     public String getName() {
          return name;
     }
```

```
public void setSendingAgent(String sa) {
     sendingagent = sa;
}

public String getSendingAgent() {
     return sendingagent;
}

public void setSendingGateway(String sg) {
     sendinggateway = sg;
}

public String getSendingGateway() {
     return sendinggateway;
}

public void prepareForTransport() throws
                 AgentTransportException {
     System.out.println("ConfidantAgent " + getName()
          + " prepareForTransport");
}

public void completedTransport() throws
                 AgentTransportException {
     System.out.println("ConfidantAgent " + getName()
           + " completedTransport");

     makeEventConnections();

     probePost();
}

public void completedItinerary() throws
                 AgentTransportException{
     System.out.println("ConfidantAgent " + getName()
           +  " completedItinerary");
}

     public void makeEventConnections() {
     System.out.println("ConfidantAgent
            makeEventHandler");
        this.makeEventHandler();
        System.out.println("ConfidantAgent
             makeEventManagerConnection");
        this.makeEventManagerConnection("localhost");
        System.out.println("ConfidantAgent
             registerAllEvents");
        this.registerAllEvents();
    }

    public void makeEventHandler() {
        try {
            super.makeEventHandler(false);
        } catch (EventHandlerException e) {
            System.out.println("Error: makeEventHandler "
                + getName());
            e.printStackTrace();
        }
    }

    public void makeEventManagerConnection(String host) {
        String url = EventManagerConnection.
           EventManagerURL(host);
        try {
            super.makeEventManagerConnection(url, false);
        } catch (NoSuchObjectException e) {
            System.out.println("Error:
                akeEventManagerConnection " + getName());
            System.out.println("NoSuchObjectException");
            e.printStackTrace();
        } catch (RemoteException e) {
            System.out.println("Error:
                makeEventManagerConnection "+ getName());
            System.out.println("RemoteException");
            e.printStackTrace();
        }
    }

    public void registerAllEvents() {
        try {
            super.registerAllEvents();
        } catch (RemoteException e) {
            System.out.println("Error: registerAllEvents "
                 + getName());
            System.out.println("RemoteException");
            e.printStackTrace();
        } catch (SecurityException e) {
            System.out.println("Error: registerAllEvents "
                  + getName());
            System.out.println("SecurityException");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Error: registerAllEvents "
                  + getName());
            System.out.println("IOException");
            e.printStackTrace();
        } catch (ServerUnavailableException e) {
            System.out.println("Error: registerAllEvents "
                 + getName());
            System.out.println("ServerUnavailableException");
            e.printStackTrace();
        }
    }

    public void probePost() {
        try {
            postEvent(getProbeEvent());
        } catch (RemoteException e) {
            System.out.println("Error: postConfidantEvent "
                 + getName());
            System.out.println("RemoteException");
            e.printStackTrace(System.out);
        } catch (SecurityException e) {
            System.out.println("Error: postConfidantEvent "
                 + getName());
            System.out.println("SecurityException");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Error: postConfidantEvent "
                 + getName());
            System.out.println("IOException");
            e.printStackTrace();
        } catch (EventException e) {
            System.out.println("Error: postConfidantEvent "
                 + getName());
            System.out.println("EventException");
            e.printStackTrace();
        } catch (ServerUnavailableException e) {
            System.out.println("Error: postConfidantEvent "
                + getName());
            System.out.println("ServerUnavailableException");
            e.printStackTrace();
        }
    }

    public void launch() {
        System.out.println("ConfidantAgent " + getName()
            + " launching");
        try {
            super.launch();
        } catch (LaunchException e) {
            System.out.println("ALARM ------ Error
                dispatching agent " + getName());
        }
    }

    public void setProbeEvent(ConfidantEvent event){
        probeevent = event;
    }

    public ConfidantEvent getProbeEvent() {
        return probeevent;
    }
}


import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
```

```
public class MD5Check {

     private MessageDigest currentAlgorithm;

     public MD5Check() {
          setAlgorithm("MD5");
     }

     public MD5Check(String md) {
          setAlgorithm(md);
     }

     private void setAlgorithm(String algorithm) {
          try {
               currentAlgorithm = MessageDigest.
                  getInstance(algorithm);
          } catch (NoSuchAlgorithmException e) {
               System.out.println("Error:
                    NoSuchAlgorithmException");
               e.printStackTrace();
          }
     }

     private byte[] loadBytes(String name) {
          FileInputStream in = null;
          ByteArrayOutputStream buffer = new
            ByteArrayOutputStream();
          int ch;
          try {
               in = new FileInputStream(name);
               while((ch = in.read()) != -1) {
                    buffer.write(ch);
               }
               in.close();
          } catch (FileNotFoundException e) {
               System.out.println("Error: loadBytes
                    FileInputStream");
               e.printStackTrace();
          } catch (IOException e1) {
               System.out.println("Error: loadBytes
                    IOException");
                e1.printStackTrace();
            }
            return buffer.toByteArray();
        }

        public byte[] computeDigest(String str) {
            currentAlgorithm.reset();
            currentAlgorithm.update(loadBytes(str));
            return currentAlgorithm.digest();
        }

        public String computeDigestAsString(String str) {
            String d = "";
            byte [] hash = computeDigest(str);
            for (int i = 0; i < hash.length; i++) {
                int v = hash[i] & 0xFF;
                if (v < 16) {
                    d += "0";
                }
                //d += Integer.toString(v, 16).toUpperCase() + " ";
                d += Integer.toString(v, 16);
            }
            return d;
        }

        public static void main(String[] args) {
            MD5Check mdc = new MD5Check();
            String str = new String();
            if (args.length == 1) {
                str = args[0];
            } else {
                str = "c:\\cldma.log";
            }
            System.out.println("Digest as byte[]");
            System.out.println(mdc.computeDigest(str));
            System.out.println("Digest as String");
            System.out.println(mdc.computeDigestAsString(str));
        }
    }
```

APPENDIX B
SUPPORTING FRAMEWORK ROUTINES FOR CONCORDIA

```
import java.io.IOException;
import java.rmi.RemoteException;

import COM.meitca.concordia.Agent;
import COM.meitca.concordia.AgentListener;
import COM.meitca.concordia.AgentTransporter;
import COM.meitca.security.InvalidLicenseException;

//the concordia server handles messaging, the ConfidantServer
//is used to provide multiple gateways on a single physical
//machine for testing purposes
public class ConfidantServer implements AgentListener {

     AgentTransporter at;

      ConfidantServer(String name) {

          System.out.println("Starting ConfidantServer: " + name );

          try {
               at = new AgentTransporter(name, 0);
          } catch (RemoteException e) {
               System.out.println("Error: AgentTransporter
                      RemoteException");
               e.printStackTrace();
          } catch (IOException e) {
               System.out.println("Error: AgentTransporter
                      IOException");
               e.printStackTrace();
          } catch (InvalidLicenseException e) {
               System.out.println("Error: AgentTransporter
                      InvalidLicenseException");
               e.printStackTrace();
          }
     }

     public void handleAgent(Agent arg0) {
          System.out.println("HandleAgent called with agent: "
                 + "Agent");
     }
     public static void main(String args[]) {

          if (args.length !=1) {
               System.out.println("Usage: ConfidantServer
                     <TransporterName>");
               System.exit(1);
          }

          ConfidantServer cs = new ConfidantServer(args[0]);

     }
}


import COM.meitca.concordia.event.*;

public class ConfidantEvent extends EventType {

     private String type;
     private String fromagent;
     private String toagent;
     private String note;

     public ConfidantEvent() {
          super("ConfidantEvent");
     }

     public ConfidantEvent(String description) {
          super(description);
     }


     /* valid event types are
      *       AgentArrived
      *       AgentArrivedACK
      *       AgentTravelRequest
      *       AgentTravelProceed
      *       MD5OK
      *       MD5Error
      *       AgentUnavailable
     *      HostUnavailable
     */
    public void setType(String t) {
         type = t;
    }
    public String getType() {
         return type;
    }

    public void setFromAgent(String fa) {
         fromagent = fa;
    }
    public String getFromAgent() {
         return fromagent;
    }

    public void setToAgent(String ta) {
         toagent = ta;
    }
    public String getToAgent() {
         return toagent;
    }


    // the note field is event-context specific
    // for instance: it is the obtained MD5 for
    // scan operations
    public void setNote(String n) {
         note = n;
    }
    public String getNote() {
         return note;
    }
}
```

LIST OF REFERENCES

[1] D. Schnackenberg, K. Djahandari, D. Sterne, Infrastructure for intrusion detection and response, DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00 2 (1999) 3–11.

[2] J. Allen, C. Alberts, S. Behrens, B. Laswell, W. Wilson, Improving the security of networked systems, Networked Systems Survivability Program, Software Engineering Institute, Carnegie Mellon University.
URL `http://www.stsc.hill.af.mil/crosstalk/2000/oct/allen.asp`

[3] J. McHugh, A. Christie, J. Allen, Defending yourself: The role of intrusion detection systems, IEEE Software 17 (5) (2000) 42–51.

[4] P. Galiasso, O. Bremer, J. Hale, S. Shenoi, D. Ferraiolo, V. Hu, Policy meditation for multi-enterprise environments, Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings (1999) 219–228.

[5] G. B. White, E. A. Fisch, U. W. Pooch, Cooperating security managers: A peer-based intrusion detection system, IEEE Network (1996) 20–23.

[6] A. J. Hoglund, K. Hatonen, A. S. Sovari, A computer host-based user anomaly detection system using the self-organizing map, Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, 2000. IJCNN 2000 5 (2000) 411–416.

[7] R. Heady, G. Luger, A. Maccabe, M. Servilla, The architecture of a network level intrusion detection system, Master's thesis, Department of Computer Science, University of New Mexico (August 1990).

[8] P. G. Neumann, P. A. Porras, Experience with EMERALD to date, in: Proceedings 1st USENIX Workshop on Intrusion Detection and Network Monitoring, 1999, pp. 73–80.
URL `http://citeseer.nj.nec.com/neumann99experience.html`

[9] G. H. Kim, E. H. Spafford, Experiences with tripwire: Using integrity checkers for intrusion detection, Tech. Rep. CSD-TR-94-012, Department of Computer Sciences, Purdue University (1994).

[10] Y. Fyodor, 'SNORTNET' - a distributed intrusion detection system (June 2000).
URL `http://snortnet.scorpions.net/snortnet.pdf`

[11] E. H. Spafford, D. Zamboni, Intrusion detection using autonomous agents, Computer Networks 34 (4) (2000) 547–570.

[12] R. Lehti, Advanced intrusion detection environment.
URL `http://www.cs.tut.fi/~rammer/aide.html`

[13] Rocksoft, Veracity - nothing can change without you knowing: Data integrity assurance.
URL `http://www.rocksoft.com/veracity/`

[14] J. McHugh, A. Christie, J. Allen, Intrusion detection: Implementation and operational issues, Software Engineering Institute, Computer Emergency Response Team/Coordination Center.
URL `http://www.stsc.hill.af.mil/crosstalk/2001/jan/mchugh.asp`

[15] S. Grafinkel, G. Spafford, A. Schwartz, Practical Unix & Internet Security, O'Reilly and Associates, 2003.

[16] C. C. Wang, Execution monitoring of security-critical programs in a distributed system: A specification based approach, Master's thesis, University of California, Davis (1996).

[17] J. D. Howard, T. A. Longstaff, A common language for computer security incidents, Sandia Report SAND98-8667, Sandia National Laboratories (October 1998).

[18] S. Kent, On the trail of intrusions into information systems, IEEE Spectrum 37 (12) (2000) 52–56.

[19] K. Seifried, LASG – attack detection.
URL `http://gd.tuwien.ac.at/opsys/linux/lasg-www/attack-detection`

[20] Tripwire for servers for security & network management, Tripwire, Inc.
URL `http://www.tripwire.com/products/servers`

[21] Tripwire manager for enterprise – wide security & network management, Tripwire, Inc.
URL `http://www.tripwire.com/products/manager`

[22] E. L. Cashin, Integrit file verification system.
URL `http://integrit.sourceforge.net`

[23] H. Debar, M. Dacier, A. Wespi, Towards a taxonomy of intrusion-detection systems, Computer Networks 31 (1999) 805–822.

[24] J. Rauch, Basic file integrity checking, SecurityFocus.com (August 2000). URL `http://www.securityfocus.com/focus/linux/articles/fileinteg.html`

[25] R. Anderson, T. Bozek, T. Logstaff, W. Meitzler, M. Skroch, K. V. Wyk, Research on mitigating the insider thread to information systems, Workshop Proceedings, RAND Corporation Report CF-163.

[26] C. Kahn, Tolerating penetrations and insider attacks by requiring independent corroboration, in: Proceedings of the 1998 Workshop on New Security Paradigms, ACM Press, 1998, pp. 122–133.

[27] S. Axelsson, The base-rate fallacy and the difficulty of intrusion detection, ACM Transactions on Information and System Security (TISSEC) 3 (3) (2000) 186–205.

[28] C. Hewitt, Viewing control structures as patterns of passing messages, Journal of Artificial Intelligence 8 (3).

[29] G. A. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, A foundation for actor computation, J. Functional Programming 7 (1).

[30] A. Fuggetta, G. P. Picco, G. Vigna, Understanding code mobility, IEEE Transactions on Software Engineering 24 (5).

[31] M. Nuttall, Survey of systems providing process or object migration, Tech Report Doc 94/10, Dept of Computing, Imperial College (May 1994).

[32] M. Crosbie, G. Spafford, Active defense of a computer system using autonomous agents, Tech. Rep. 95-008, COAST Group, Department of Computer Sciences, Purdue University (February 1995).

[33] J. S. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. Spafford, D. Zamboni, An architecture for intrusion detection using autonomous agents, Proceedings. 14th Annual Computer Security Applications Conference (1998) 13–24.

[34] M. C. Bernardes, E. dos Santos Moreira, Implementation of an intrusion detection system based on mobile agents, Proceedings. International Symposium on Software Engineering for Parallel and Distributed Systems (2000) 158–164.

[35] W. Jansen, P. Mell, T. Karygiannis, D. Marks, Applying mobile agents to intrusion detection and response, National Institute of Standards and Technology, Computer Security Division (1999). URL `http://csrc.nist.gov/publications/nistir/ ir6416.pdf`

[36] C. Coosta, Tach design concept, Lockeed Martin Corporation.

[37] G. Wang, Moible agent file integrity analyzer, Master's thesis, University of Central Florida (2001).

[38] R. F. DeMara, A. J. Rocke, Mitigation of network tampering using dynamic dispatch of mobile agents, Computers & Security 23 (1) (2004) 31 – 42.

[39] T. Bott, Evaluating the risk of industrial espionage, in: Proceedings of the Reliability and Maintainability Symposium, 1999, pp. 230–237.

[40] R. Anderson, M. Kuhn, Tamper Resistance - a Cautionary Note, in: Proceedings of the Second Usenix Workshop on Electronic Commerce, 1996, pp. 1–11. URL `http://citeseer.nj.nec.com/anderson96tamper.html`

[41] File integrity checkers. URL `http://www.networkintrusion.co.uk/integrity.htm`

[42] T. Linden, Nabou system integrity monitor. URL `http://www.nabou.org/en/software/nabou/`

[43] WetStone Technologies, Inc., SMART Watch. URL `http://www.wetstonetech.com/smartwatch_ns.html`

[44] C. Kruegel, T. Toth, Applying mobile agent technology to intrusion detection, in: ICSE Workshop on Software Engineering and Mobility, 2001. URL `http://citeseer.nj.nec.com/kr01applying.html`

[45] Y. Kun, G. Xin, L. Dayou, Security in mobile agent system: Problems and approaches, ACM SIGOPS Operating Systems Review 34 (1) (2000) 21–28.

[46] G. Helmer, J. Wong, V. Honavar, L. Miller, Intelligent agents for intrusion detection (September 1998). URL `http://citeseer.nj.nec.com/helmer98intelligent.html`

[47] D. A. Frincke, D. Tobin, J. C. McConnell, J. Marconi, D. Polla, A framework for cooperative intrusion detection, in: Proc. 21st NIST-NCSC National Information Systems Security Conference, 1998, pp. 361–373. URL `http://citeseer.nj.nec.com/frincke98framework.html`

[48] IBM Research, Aglets. URL `http://www.trl.ibm.com/aglets`

[49] J. Lu, Mobile agent protocols for distributed detection of network intrusions, Master's thesis, University of Central Florida (2000).

[50] Y. Zhu, Decentralized control schemes for coordinating distributed processing activities of mobile software agents, Master's thesis, University of Central Florida (2000).

[51] B. Kapoor, Remote misuse detection system using mobile agents and relational database query techniques, Master's thesis, University of Central Florida (2000).

[52] M. J. Ranum, Experiences benchmarking intrusion detection systems, Tech. rep., NFR Security (December 2001).
URL `http://www.nfr.com/forum/white-papers/Benchmarking-IDS-NFR.pdf`

[53] G. Fink, B. Chappell, T. Turner, K. O'Donoghue, A metrics-based approach to intrusion detection system evaluation for distributed real-time systems, in: International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops, 2002.

[54] T. Champion, M. L. Denz, A benchmark evaluation of network intrusion detection systems, in: Proceedings of the IEEE Conference on Aerospace Systems, 2001.

[55] D. J. Bodeau, Information assurance assessment: Lessons-learned and challenges, in: Proceedings of the ACSA Workshop on Information Security System Scoring and Ranking, 2001.

[56] D. J. Marchette, Computer Intrusion Detection and Network Monitoring – A Statistical Viewpoint, Springer, 2001.

[57] S. Northcutt, J. Novak, Network Intrusion Detection, 3rd Edition, New Riders, 2003.

[58] S. Axelsson, The base-rate fallacy and its implications for the difficulty of intrusion detection, in: Proceedings of the 6th ACM Conference on Computer and Communications Security, 1999, pp. 1–7.

[59] W. Mendenhall, T. Sincich, Statistics for Engineering and the Sciences, Dellen Publishing Company, 1992.

[60] M. H. Zweig, G. Campbell, Receiver-operating characteristic (roc) plots: A fundamental evaluation tool in clinical medicine, Clinical Chemistry 39 (4) (1993) 561–577.

[61] ROC curve analysis: Introduction.
URL `http://www.medcalc.be/manual/mpage06-13a.php`

[62] A. J. Rocke, R. F. DeMara, CONFIDANT: Collaborative object notification framework for insider defense using autonomous network transactions, submitted to Journal of Autonomous Agents and Multi-Agent Systems.

[63] P. Kotzanikolaou, M. Burmester, V. Chrissikopoulos, Secure transactions with mobile agents in hostile environments, in: Australasian Conference on Information Security and Privacy, 2000, pp. 289 – 297.

[64] W. Jansen, Countermeasures for mobile agent security, National Institude of Standards and Technology.
URL `http://csrc.nist.gov/mobilesecurity/Publications/ppcounterMeas.pdf`

[65] W. Jansen, T. Karygiannis, NIST special publication 800-19 – mobile agent security, National Institute of Standards and Technology.
URL `http://csrc.ncsl.nist.gov/mobilesecurity/Publications/sp800-19.pdf`

[66] S. R. Schach, Classical and Object-Oriented Software Engineering with UML and C++, 4th Edition, WCB McGraw-Hill, 1999.

[67] Mobile agent computing, Mitsubishi Electric ITA Horizon Systems Laboratory (January 1998).
URL `http://www.meitca.com/HSL/Projects/Concordia/MobileAgentsWhitePaper.pdf`

[68] J. Wack, M. Tracy, M. Souppaya, Guideline on network security testing, National Institute of Standards and Technology, Computer Security Division (2003).
URL `http://csrc.ncsl.nist.gov/publications/nistpubs/800-42/NIST-SP800-42.pdf`

[69] CERIAS - autonomous agents for intrusion detection.
URL `http://www.cerias.purdue.edu/about/history/coast/projects/aafid.php`

[70] A. Waterland, Stress: impose a configurable amount of computer system load.
URL `http://weather.ou.edu/ apw/projects/stress/`

[71] OpenSSL: Open source toolkit implementing the secure socket layer protocol.
URL `http://www.openssl.org/`

[72] P. Burkholder, Ssl man-in-the-middle attacks, Tech. rep., SANS (2002).
URL `http://www.sans.org/rr/papers/60/480.pdf`

[73] IBM PCI cryptographic coprocessor.
URL `http://www-3.ibm.com/security/cryptocards/html/library.shtml`