Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2007-023 April 10, 2007

Self-Adaptive Systems for Information Survivability: PMOP and AWDRAT

Howard Shrobe, Robert Laddaga, Robert Balzer, Neil Goldman, Dave Wile , Marcelo Tallis, Tim Hollebeek, and Alexander Egyed

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Self-Adaptive Systems for Information Survivability: PMOP and AWDRAT

Howard Shrobe
MIT CSAIL
mule 32 Vassar Street
Cambridge, MA 02139

Robert Laddaga
BBN Technologies
10 Moulton Street
Cambridge, MA 02139

Bob Balzer, Neil Goldman
Dave Wile, Marcelo Tallis
Tim Hollebeek, Alexander Egyed
Teknowledge
4640 Admiralty Way, Suite 1010
Marina del Rey, CA 90292

Abstract

Information systems form the backbones of the critical infrastructures of modern societies. Unfortunately, these systems are highly vulnerable to attacks that can result in enormous damage. Furthermore, traditional approaches to information security have not provided all the protections necessary to defeat and recover from a concerted attack; in particular, they are largely irrelevant to the problem of defending against attacks launched by insiders.

This paper describes two related systems PMOP and AWDRAT[1] *that were developed during the DARPA Self Regenerative Systems program. PMOP defends against insider attacks while AWDRAT is intended to detect compromises to software systems. Both rely on self-monitoring, diagnosis and self-adaptation. We describe both systems and show the results of experiments with each.*

1 Background and Motivation

The infrastructure of modern society is controlled by computational systems that are vulnerable to information attacks that can lead to consequences as dire as those of modern warfare. In virtually every exercise conducted by the government, the attacking team has compromised the target systems with little difficulty. Hence, there is an urgent need for new approaches to protect the computational infrastructure from such attacks and to enable it to continue functioning even when attacks have been successfully launched.

[1]This article describe research conducted at the Computer Science and Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided by the Information Processing Technology Office of the Defense Advanced Research Projects Agency (DARPA) under AFRL Contract Numbers FA8750-04-2-0240 and FA8750-04-C-0252 The views presented are those of the author alone and do not represent the view of DARPA or AFRL.

Our premise is that to protect this infrastructure we need to restructure its software systems as *Self Adaptive Survivable Systems*. In particular, we believe that a software system must be capable of detecting its own malfunction, it must be capable of diagnosing the cause of the problem, it must be capable of repairing itself, and it must be capable of preventing itself from doing harm, even if it is compromised.

Our work is set in the difficult context in which there is a concerted and coordinated attack by a determined adversary who may be either an external attacker or a privileged insider. This context places an extra burden on the self-protective machinery. It is no longer adequate merely to determine which component of a computation has failed to achieve its goal, in addition we wish to determine whether that failure is indicative of a *compromise* to the underlying infrastructure and whether it is due to an insider abusing his privileges. In addition, we need to assess whether the problem is likely to lead to failures of other computations at other times. Finally, we need to regenerate the system so that it can continue its work with reduced threat and lower likelihood of doing harm.

This paper describes 2 related systems, PMOP and AWDRAT that were developed as part of the DARPA Self Regenerative Systems program. These two systems share a base of common tools and were applied to the defense of a common test system. However, PMOP's focus is detection of insider attacks, while AWDRAT's is on external threats. PMOP's job is to determine that the operator has asked the system to do something that will lead to harm (whether or not the system is compromised), while AWDRAT's task is to determine that the system has behaved incorrectly in response to a legitimate request. Both share a common philosophy that the system must monitor its own behavior, have models of itself that it can reason about and adapt itself to prevent harm from occurring.

Both PMOP and AWDRAT are model based and detect differences from expected behavior. AWDRAT detects

differences from predicted behavior (what the system was contracted to do) and PMOP detects differences from benign behavior (what the larger system - comprising the user and the application - was designed to do). AWDRAT detects when the *application* has been compromised while the PMOP detects when the *user* has been compromised.

1.1 A Common Monitoring Infrastructure

The common infrastructure shared by these systems is a self-monitoring framework consisting of:

- **Wrappers** that are placed around critical parts of the system, to collect data and control whether and how application level operations are performed.
- **An Architectural System Model** that is capable of predicting how the system ought to behave in response to user-level and internal requests. It is fine-grained enough to provide an architectural level view of how the system operates, but coarse grained enough to avoid excessive overhead.

Both the AWDRAT and PMOP architectures are intended to be widely applicable and could, in principle, be applied to a variety of target systems written in a variety of programming languages for use on a variety of platforms; however, wrapper technologies are often specific to a particular programming language and/or operating system environment. Our current set of wrapper technologies limits us to Java and C programs running in a Windows environment.

PMOP and AWDRAT employ two distinct wrapper technologies: SafeFamily[1, 4] and JavaWrap. The first of these encapsulates system DLL's, allowing AWDRAT and PMOP to monitor accesses to external resources such as files or communication ports. To use the SafeFamily facility, one must provide an XML file of rules specifying the resources (e.g. files, ports) and actions (e.g. writing the file, communicating over the port) that are to be prevented.

The second wrapper facility provides method wrappers for Java programs, delivering a capability similar to ":around" methods in the Common-Lisp Object System[5, 2] or in Aspect-J[6]. To use the JavaWrap facility, one must provide an XML file specifying the methods one wants to wrap as well as a Java Class of mediator methods, one for each wrapped method in the original application. When a class-file is loaded, JavaWrap rewrites the wrapped methods to call the corresponding wrapper methods; wrapper methods are passed a handle to the original method allowing them to invoke the original method if desired.

These two capabilities are complementary: JavaWrap provides visibility to all application level code, SafeFamily provides visibility to operations that take place below the abstraction barrier of the Java language runtime model. Together they provide AWDRAT with the ability to monitor the applications behavior in detail as is shown in Figure 1.

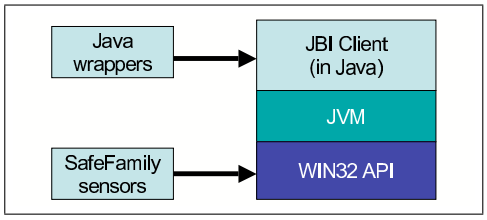

Figure 1. Two Types of Wrappers Used in PMOP and AWDRAT

2 PMOP: Protecting Against Insider Threats

PMOP (Prevention of Misuse of Operator Privileges) is concerned with threats from insiders. Insiders are distinguished from other attackers by the fact that they have been granted access to the system being defended, have been granted privileges on that system, and know how it operates. This means that we must assume that the insider has all the access, privileges, and knowledge needed to effect an attack and that traditional security mechanisms, which are focused on detecting and preventing attempts to escalate the user's privileges, are ineffective. Instead the PMOP project focuses on detecting the application behavior that will cause harm in the real world.

2.1 The PMOP Architecture

The job of the PMOP architecture is to detect and prevent actions that could lead to harm. The components that comprise the overall PMOP architecture are shown in figure 2. Wrappers intercept operator requests issued to the running application and forward the stream of requests to the operational system model which makes predictions about the effects of the actions that the system is asked to perform. The predictions of the operational system model are then assessed for whether they would lead to harmful effects. If so, then a further assessment is made as to the likelihood that the harm was intended. If it appears so then the system increases its degree of suspicion of the user (and notifies human security personnel); if not, then the actions are deemed to be an operator error and feedback is provided to the user. If the user's request is deemed to be legitimate (i.e. causes no harm) then the intercepted operator request is passed to the application to be processed.

A level of suspicion is established by the relative degree to which the user's actions fit the role-based plans to the exclusion of the attack plans. This level of suspicion triggers unique effectors that contain the effects of suspected

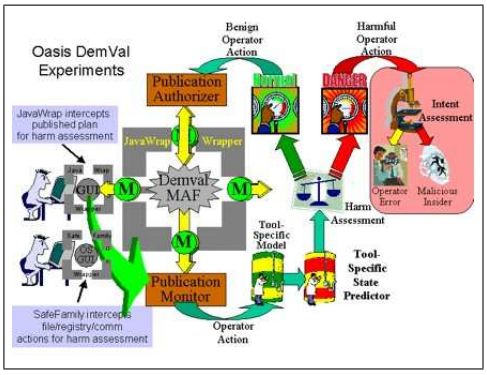

Figure 2. The PMOP Architecture

insider attacks (in a dynamic process-level virtual machine) to protect the system while additional evidence is gathered; administrators can determine whether to authorize or quarantine the contained actions.

Two unique capabilities result from detecting attacks based on model-based predicted harm (about to be) caused to a system:

1. There is no need to update the defense as new insider attacks are discovered or new ways to obfuscate them are invented.
2. Attacks based on corrupted operand values or the situation in which operations are invoked can be detected and blocked.

Because our attack detectors are model based, and thus harder to fool, the vast majority of insider attacks can be detected and blocked (at least to the level of sophistication of the system models and harm inference reasoning).

The assumption that the insider has all the access, privileges, and knowledge needed for an attack - which defines what it means to be an insider - means that insider attack detection and thwarting must be based on the attack behavior itself. Although detection could occur after the fact, from observation of the damage caused, thwarting requires the detection to occur before the damage has been caused so that it can be prevented.

The detection must therefore be based on pre-damage activity - namely the user's commands or directives to some software system and that software's execution of system level operations to effect those commands or directives. Neither of these phenomena is normally available, but can be made so through proper instrumentation (discussed below).

2.2 Operational System Model

These additional capabilities do not come for free. They are based on the availability of an individually constructed operational system model, whose quality is determined by the fidelity of that model. In the example system studied in this project, the models were initially propositional rule bases, from which we inferred both the predicted state of the system and the likelihood of harm resulting from the change of state. In addition to modeling the system in its nominal state, one could also model the system in various states of compromise. For example, if a system has had its effective communication bandwidth reduced by a network denial of service attack, we need to infer the effect of user actions in that context, not simply the nominal context. In our earliest efforts the models were only able to detect harm when that harm was explicitly associated with the predicted state (e.g. detonating a missile before launching it) or it could be inferred from the transition into the predicted state on the basis of one of the following transition rules:

1. Actions that make resources unavailable to authorized users;
2. Actions that make resources available to unauthorized users;
3. Actions that inject disinformation into databases;
4. Actions that delete truthful information from databases.

More subtle attacks, as for example, injecting truthful but misleading information into a database, will have to wait for more refined operational system models and more powerful harm-inference reasoners.

2.3 Behavior Monitor (Sensors)

Detection is based on a set of application-level sensors that are able to detect user actions as they occur.

Wrappers monitor in real-time the users actions in a legacy application, including typing, editing, copying, pasting, and other actions. Actions are sensed uniformly whether they are invoked via menus, shortcuts, toolbars, or mouse-dragging. Relevant parameter values like the text that was entered, copied, or deleted are also accessible.

Conceptually, our wrapper is positioned between the applications high-level user interface and the application itself, allowing the wrapper to monitor the users interaction with the application in terms of the applications own object model. This application model is defined by and accessed through a COM API that allows external code to query and manipulate this model (i.e. the application state).

This COM API and the application object model to which it provides access are the heart of our approach to providing an application-level behavior monitor. The API defines the application-level operations that the application can perform (whether invoked through the GUI or through script) and the operands needed for those operations.

2.4 Impending Harm Detector

By intercepting and mediating the calls from the GUI on these COM APIs, or the internal methods behind the COM API, the user's application-level actions can be captured and screened before those judged safe are allowed to pass onto the application itself for processing. By mediating the communication between the GUI and the application, the application-level actions are directly accessible (because they are the operations that the application is capable of performing). The GUI has performed the idiosyncratic translation from interface gestures (keyboard input, mouse clicks, and drags) to these application-level actions. We are merely mediating its communication with the application once this translation has occurred.

2.5 Malicious Behavior Detector

We developed a malicious behavior detector based on data received from these wrappers that analyzes the application-level user modification history relative to a role-based model of expected behavior. This model identifies both the types of behavior expected in a situation and the means for assessing the appropriateness of the particular behavior observed. The assessment uses a variety of mechanisms for determining the appropriateness of an action such as safety models, plant models, design rules, best practices, and heuristics. This analyzer detects both intentional and accidental actions that harm the system.

The suspicious behavior detector differentiates the two by inferring user goals from the observed harmful behavior, recent historical behaviors, and the set of plans consistent with the larger behavior context. These plans are extracted from a library and include both plans associated with the users role and attack plans (both generic and site specific). A level of suspicion is established by the relative degree to which the users actions fit the role-based plans to the exclusion of the attack plans.

This library plays no part in the detection of an attack (based solely on predicted harm). Instead it is used after an attack has been detected to distinguish malicious intent (following an attack plan) from inadvertent operator error (following a role-based expected behavior plan).

2.6 Intent Inference

Inferring the actual intent of an operator is a very difficult task. In many cases, an operator could perform a harmful action either by accident or intentionally. To facilitate this, we log all user actions, to call attention to those that do eventually cause harm, and for those that seem malicious, to open a case book on the user, documenting the suspicious behavior and leaving final determinations of intent to human examiners.

However, some behavior is more suspicious than others and so part of our goal is to identify those actions that are more likely to have been the result of malicious intent. We have developed guiding principles for this assessment: A set of actions that is consistent with a plan for causing harm but not consistent with normal operations is cause for suspicion. The larger the deviation between the two (e.g. the number of steps consistent with harmful outcome, but inconsistent with benign outcome) is a metric of how suspicious the activity is.

We use Computational Vulnerability Analysis [12] to develop a library of abstract attack plans that an insider might use to render harm. Similarly we provide a library of abstract normal (or role-based) plans that are consistent with normal operator behavior. Using plan recognition techniques, we then assess how well the logged behavior matches any of these plans and then accordingly score the operator actions.

2.7 Demonstration System

To demonstrate the applicability of our Misuse Prevention architecture to legacy systems, we chose a moderately large example legacy system to model and defend against insider attacks, the OASIS Dem/Val system, developed under an earlier DARPA program. This system relies on the Joint Battlespace Infosphere (JBI) repository and communication protocol for coordinating and managing information from a variety of agents cooperating in the development of major military plans. The OASIS Dem/Val system developed air tasking orders for air cargo and munitions delivery and deployment and was created to demonstrate how existing military systems could interoperate with new components through the JBI infrastructure. In particular, we focused mainly on a single application within OASIS Dem/Val, the MAF/CAF mission planner, an interactive, Java-based, graphical editor for producing flight plans.

PMOP used its two types of wrappers as follows: Java wrappers are used to derive state by intercepting calls to publish and read information from the repository. The attempts to publish are filtered by our Harm Detector and Malicious Behavior Detector; if they determine that harm will ensue, the publication is blocked. SafeFamily wrappers are used to monitor all accesses to local resources, such as files, communications ports, etc. Information gathered fro these wrappers are used to detect and prevent harm to these resources.

The operational model we developed for the OASIS system was used to detect corrupted data, effectively data whose use in the final air tasking order would have harmed the mission. The model itself is expressed as a set of rules

that constitute the application semantics.

The following sample rules are typical of those that determine whether an air tasking order produced by the MAF / CAF operators is harmful.

- Planes have types, which have a maximum Range before the plane must land or be refueled (refueling resets the starting point to the refueling point i.e. assumes the plane has been fully refueled).
- Planes have types which have a minimum required runway length for takeoffs and landings
- Planes cannot land or takeoff in restricted-access zones.
- Refueling can only occur in designated refueling areas.
- A plane's weight (determined by its plane type) cannot exceed the weight-handling maximum for each runway it lands on or takes off from.
- A plane can only land or take off from a runway at night (1800 to 0600 local time) if that runway is equipped with night lighting.
- The duration of a leg must exceed the time needed to fly that leg (i.e. the distance between its start and end locations) at the plane's maximum speed

The rules monitor an operator's behavior to detect harm at the point that the operator's actions are committed. In our case, this is when the Mission Plan constructed by the operator is published. Harm is detected by determining whether the plan satisfies the integrity constraints illustrated above. If not, its publication is blocked to prevent that harm. An analysis is then performed on the offending action, the failed integrity check(s), and the history of operator actions that led to the offending action to determine whether there is a consistent pattern of malicious operator activity. Harmful plans are characterized using a relatively simple rule-based inferencing system and are then archived in a "case-file" that stores and compares several bad plans produced by the same operator.

2.8 Validation: Red Team Experiment

In order to validate our system's ability to detect and thwart insider attacks either through the application's GUI or through the operating system GUI (the Explorer process), a Red Team experiment was conducted. Out of 14 attempts to harm the application or induce a false positive, no attacks causing harm and one false positive were induced through the application's GUI, while one attack causing harm and no false positives were induced through the operating system GUI.

3 AWDRAT : Protecting Against External Attacks

Like PMOP, AWDRAT [13] is a middleware system to which an existing application software system (the "target system") may be retrofitted. AWDRAT provides immunity to compromises of the target system, making it appear self-aware and capable of actively checking that its behavior corresponds to that intended by its designers. "AWDRAT" stands for Architectural-differencing, Wrappers, Diagnosis, Recovery, Adaptivity and Trust-modeling.

AWDRAT uses these facilities in order to provide the target system with a cluster of services that are normally taken care of in an *ad hoc* manner in each individual application, if at all. These services include fault containment, execution monitoring, diagnosis, recovery from failure and adaption to variations in the trustworthiness of the available resources. Software systems tethered to the AWDRAT environment behave adaptively; furthermore, with the aid of AWDRAT, these system regenerate themselves when attacks cause serious damage.

3.1 The AWDRAT Approach

Before delving into the details, it's useful to understand the general approach taken by AWDRAT. AWDRAT can be applied to a "legacy" system, without modifying the source code of that system. Instead, the programmer provides AWDRAT with a "System Architectural Model" that specifies how the program is intended to behave; usually this description is provided at a fairly high level of abstraction (this model can be thought of as an "executable specification" of the target system). AWDRAT checks that the actual behavior of the target system is consistent with that specified in the System Architectural Model. If the actual behavior ever diverges from that specified in the model, then AWDRAT suspends the program's execution and attempts to diagnose why the program failed to behave as expected. The diagnostic process identifies an attack and a set of resources (e.g. binary code in memory, files, databases) that might have been corrupted by the attack together with a causal chain of how the attack corrupted the resources and of how the corruption of the resources led to the observed misbehavior. AWDRAT then attempts to repair the corrupted resources if possible (for example by using backup copies of the resources). Finally AWDRAT restarts the application from the point of failure and attempts to find a way to continue rendering services without using resources that it suspects might still be compromised.

3.2 The AWDRAT Architecture

The AWDRAT architecture is shown in Figure 3. This architecture includes both a variety of models maintained by AWDRAT (the round boxes in the figure) as well as a number of major computational components (the square boxes in the figure). AWDRAT is provided with a model of the intended behavior of the target system (the System Architectural Model in the figure). AWDRAT actively monitors the actual behavior of the target system using "wrapper" technology. The Wrapper Synthesis module in the figure is responsible for synthesizing non-bypassable wrappers (shown in the figure surrounding the target system). These wrappers instrument the target system and deliver observations of its behavior to the component labeled "Architectural Differencer". This module consults the System Architectural Model and check that the observations of the target system's behavior are consistent with the prediction of the System Architectural Model suspending the target system's execution if they are inconsistent. In the event that unanticipated behavior is detected, a description of the discrepancy between expected and actual behaviors is send to the AWDRAT component labeled Diagnosis in the figure. The AWDRAT Diagnosis module uses Model-Based Diagnosis to determine the possible ways in which the system could have been compromised so as to produce the observed discrepancy. AWDRAT proceeds to use the results of the diagnosis to calculate the types of compromise that may have effected each computational resource of the target system. AWDRAT also calculates the likelihood of each possible compromise. These results are stored in an internal model, labeled "Trust Model" in the figure.

Next, AWDRAT attempts to help the target system recover from the failure. First it uses backup and redundant data to attempt to repair any compromised resources (the component labeled Recover and Regeneration in the figure) and then, during and after recovery, AWDRAT tries to help the target avoid using any residual compromised resources by using alternative methods that are capable of achieving the target system's goals and that don't require the use of the compromised resources. The module in the figure labeled "Alternative Method Selection" is responsible for choosing between such alternative methods using decision theoretic techniques. Similarly, even if there is only one possible method for a task, there is often the possibility of choosing between alternative resources (e.g. there might be redundant copies of data, there might be the possibility of running the code on more than one host). Such choices are also managed by the Alternative Method Selection component of AWDRAT. Part of the reasoning involved in making these choices is guided by the Trust Model: If a resource is potentially compromised then there is a possibility that any method using it will lead to a system failure. However, some methods might be much more desirable than others because they deliver better quality of service (e.g. because they run faster, or render better images). The method selection module, therefore, attempts to find a combination of method and resources that makes a good tradeoff, maximizing the quality of service rendered and minimizing the risk of system failure.

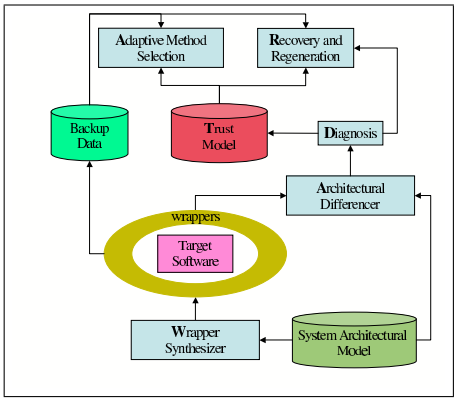

Figure 3. The AWDRAT Architecture

AWDRAT also uses the target system's System Architectural Model to recognize the critical data that must be preserved in case of failure. AWDRAT's Wrapper Synthesizer module generates wrappers that dynamically provision backup copies and redundant encodings of this critical data (labeled Backup Data in the figure). During recovery efforts, AWDRAT uses these backup copies to repair compromised data resources; in addition, the AWDRAT Adaptive Method Selection module may decide to use the backup copies of data instead of the primary copy.

Using this combination of technologies, AWDRAT provides "cognitive immunity" to both intentional and accidental compromises. An application that runs within the AWDRAT environment appears to be self-aware, knowing its plans and goals; it actively checks that its behavior is consistent with its goals and provisions resources for recovery from future failures. AWDRAT builds a "Trust Model" shared by all application software, indicating which resources can be relied on for which purposes. This allows an application to make rational choices about how to achieve its goals.

3.3 Synthesis of Wrappers and Execution Monitor

AWDRAT uses the same wrapper frameworks as PMOP; the inputs to the wrapper facilities (the JavaWrap XML spec, the Java Mediator files and the SafeFamily XML specification file) are not written by the user, but are instead automatically generated by AWDRAT from its "System Architectural Model". The System Architectural Model is

written in a language similar to the "Plan Calculus" of the Programmer's Apprentice [9, 10, 8]; it includes a hierarchical nesting of components, each with input and output ports connected by data and control-flow links. Each component is provided with prerequisite and post-conditions. In AWDRAT, we have extended this notation to include a variety of event specifications, where events include the entry to a method in the application, exit from a method or the attempt to perform an operation on an external resource (e.g. write to a file). The occurrence of an entry (exit) event indicates that a Java method corresponding to a component in the System Architectural Model has started (completed) execution; these events are generated by JavaWrap. A prohibited event occurs when the target attempts to access a resource in a way not sanctioned by the System Architectural Model; this is detected by SafeFamily. Similarly invocation of a Java method not predicted by the System architectural model is treated as a prohibited event; this is detected by JavaWrap.

Given this information, the AWDRAT wrapper synthesizer collects up all event specifications used in the System Architectural Model and then synthesizes the wrapper method code and the two required XML specification files.

3.4 Architectural Differencing

In addition to synthesizing wrappers, the AWDRAT generator also synthesizes an "execution monitor" corresponding to the System Architectural Model The role of the wrappers is to create an "event stream" tracing the execution of the application. The role of the execution monitor is to interpret the event stream against the specification of the System Architectural Model and to detect any differences between the two as shown in Figure 4. Should a deviation be detected, diagnosis and recovery is attempted.

The System Architectural Model provided to AWDRAT includes prerequisite and post-conditions for each of its components. A special subset of the predicates used to describe these conditions are built into AWDRAT and provide a simple abstract representation of data structuring. The AWDRAT synthesizer analyzes these statements and generates code in the Lisp mediators that creates backup copies of those data-structures which are manipulated by the application and that the System Architectural Model indicates are crucial.

Using these generated capabilities, AWDRAT detects any deviation of the application from the abstract behavior specified in its System Architectural Model and invokes its diagnostic services.

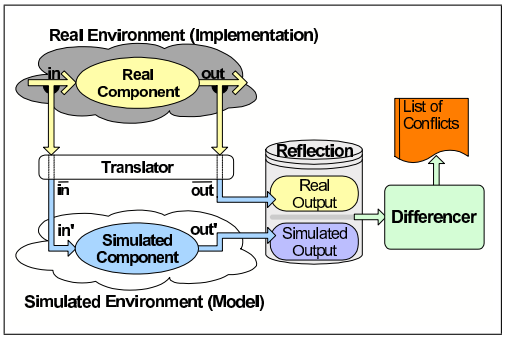

Figure 4. Architectural Differencing

3.5 Diagnostic Reasoning

AWDRAT's diagnostic service is described in more detail in [11] and draws heavily on ideas in [3]. Each component in the System Architectural Model provided to AWDRAT is provided with behavioral specifications for both its normal mode of behavior as well as additional specifications of known or anticipated faulty behavior. As explained in section 3.4, an event stream tracing the execution of the application system, is passed to the execution monitor, which in turn checks that these events are consistent with the System Architectural Model. The execution monitor builds up a data base of assertions describing the system's execution and connects these assertions in a dependency network. Any directly observed condition is justified as a "premise" while those assertions derived by inference are linked by justifications to the assertions they depend upon. In particular, post-conditions of any component are justified as depending on the assumption that the component has executed normally as is shown in Figure 5. This is similar to the reasoning techniques in [10].

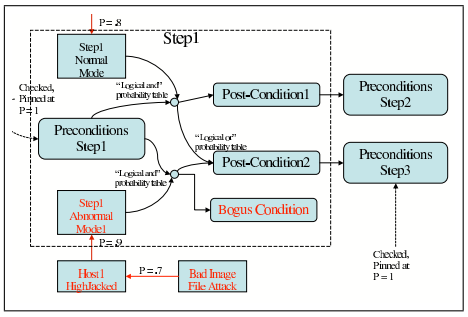

Figure 5. Dependency Graph

Should a discrepancy between actual and intended behavior be detected, it will show up as a contradiction in the database of assertions describing the application's execution history. Diagnosis then consists of finding alternative behavior specifications for some subset of the components

in the System Architectural Model such that the contradiction disappears when these specifications of off-nominal behavior are substituted.

In addition to modeling the behavior of the components in the System Architectural Model, AWDRAT also models the health status of resources used by the application. We use the term "resource" quite generally to include data read by the application, loadable files (e.g. Class files) and even the binary representation of the code in memory. Part of the System Architectural Model provided to AWDRAT describes how a compromise to a resource might result in an abnormal behavior in a component of the computation; these are provided as conditional probability links. Similarly, AWDRAT's general knowledge base contains descriptions of how various types of attacks might result in compromises to the resources used by the application as is shown in Figure 6. AWDRAT's diagnostic service uses this probabilistic information as well as the symbolic information in the dependency network to build a Bayesian Network and thereby to deduce the probabilities that specific resources used by the application have been compromised.

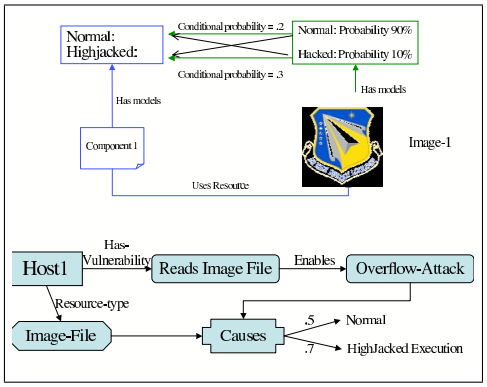

Figure 6. Diagnosis With Fault and Attack Models

3.6 Self-Adaptive Software

Recovery in AWDRAT depends critically on self-adaptive techniques such as those described in [7]. The critical idea is that in many cases an application may have more than one way to perform a task. Self-adaptive software involves making dynamic choices between such alternative methods.

The general framework starts from the observation that we can regard alternative methods as different means for achieving the same goal. But the choice between methods will result in different values of the "non-functional properties" of the goal; for example, different methods for loading images have different speeds and different resulting image quality. The application designer presumably has some preferences over these properties and we have developed techniques for turning these preferences into a utility function representing the benefit to the application of achieving the goal with a specific set of non-functional properties. Each alternative method also requires a set of resources (and these resources must meet a set of requirements peculiar to the method); we may think about these resources having a cost. As is shown in Figure 7, the task of AWDRAT's adaptive software facility is to pick that method and set of resources that will deliver the highest net benefit. Thus AWDRAT's self-adaptive software service provides a decision theoretic framework for choosing between alternative methods.

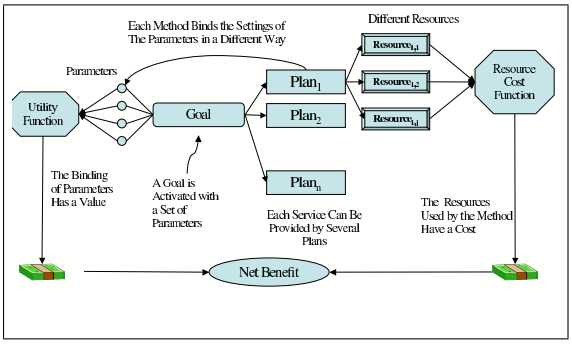

Figure 7. Adaptive Software Picks the Best Method

3.7 Recovery and Trust Modeling

As shown in Figure 8, the results of diagnosis are left in a Trust Model that persists beyond the lifetime of a particular invocation of the application system. This Trust Model contains assessments of whether system resources have been compromised and with what likelihood. The Trust Model guides the recovery process.

Recovery consists of first resetting the application system to a consistent state and then attempting to complete the computation successfully. This is guided by the Trust Model and the use of self-adaptive software. One form of recovery, for example, consists of restarting the application and then rebuilding the application state using resources that are trustable. This consists of:

- Restarting the application or dynamically reloading its code files (assuming that the application system's language and run-time environment supports dynamic loading, as does Java or Lisp, for example). In doing so AWSDRAT uses alternative copies of the loadable code files if the Trust Model indicates that the primary

copies of the code files have possibly been compromised.
- Using alternative methods for manipulating complex data, such as image files or using alternative copies of the data resources. The idea is to avoid the use of resources that are likely to have been compromised.
- Rebuilding the application's data structures from backup copies maintained by the AWDRAT infrastructure.

The Trust Model enters into AWDRAT's self-adaptive software infrastructure by extending the decision theoretic framework to (1) Recognize the possibility that a particular choice of method might fail and to (2) associate a cost with the method's failure (e.g. the cost of information leakage). Thus, the expected benefit of a method is the raw benefit multiplied by the probability that the method will succeed while the cost of the method includes the cost of the resources used by the method plus the cost of method failure multiplied by the probability that the method will fail (i.e. expected cost). The probability of success is just the joint probability that all required resources are in their uncompromised states (and the failure probability is just 1 minus the probability of success). In decision theoretic terms, the best method is, in this revised view, the one with the highest net expected benefit. This approach allows AWDRAT to balance off the attraction of a method that provides a highly desirable quality of service against the risk of using resources that might be compromised.

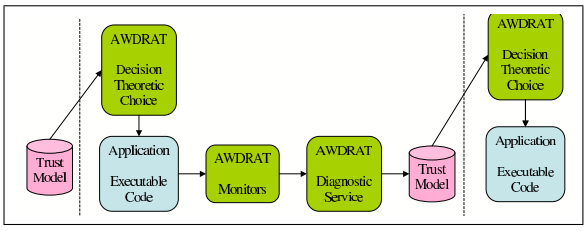

Figure 8. The Trust Model Guides Future Recovery

3.8 Experimentation and Results

AWDRAT's goal is to guarantee that the target system tethered to it faithfully executes the intent of the software designer. To assess AWDRAT we applied it to the defense of MAF interactive mission planner, also used as the target system for the PMOP experiments. We identified the following properties that AWDRAT should guarantee:

- The data structures maintained by the application should accurately reflect user requests:
- The application should not be able to open a port to any application other than to those JBI servers that it must communicate with (i.e. the JBOSS server).
- The application should not be able to open a port to a foreign host
- The application should not be able to write files except in the specified set of directories that constitute its temporary storage areas.

These properties are rather typical of the security concerns that one would have for any program, namely that the integrity of its data structures should be maintained, that it shouldn't divulge internal information except as authorized and that it shouldn't modify external information.

To test the effectiveness of AWDRAT in guaranteeing these properties we adopted three distinct ways of attempting to violate the properties:

- Modification of class files. We hand inserted attack code into a Java source file, recompiled and placed the class file into the appropriate directory on the Java Classpath. This is intended to simulate the case where an external attacker gained sufficient privileges to corrupt the class files.
- Random attacks. The AWDRAT controller launches a parasitic thread on startup. This thread waits for a random delay and then attempts to violate one of the constraints listed above (e.g. attempts to launch an application, to open a port, or to corrupt the data structures). Which type of violation (and in the case of data integrity which field of which data structure) is chosen randomly; the specific attack selected is logged for analysis purposes.
- Wrapped methods. AWDRAT places wrappers around a significant number of methods in the MAF application. The wrappers can be used as a place from which to launch a simulated attack; for example, by wrapping the "loadImage" method, one can simulate an attack payload that is carried by a corrupted image file (without actually having to do the very laborious work of constructing such a corrupted image file).

These should be thought of as different ways of introducing unexpected behavior into the MAF program; they do not correspond directly to any particular attacks. Rather they correspond more closely to the effects that a variety of different attacks might have in corrupting files used by the target system or in modifying its binary code in memory. We observe that for an attack to be effective it must cause the target system to:

1. divulge information that it is not intended to
2. modify information that it is not intended to
3. modify its own state in ways that are not sanctioned by its specification

4. consume resources that starve out other applications
5. fail to meet its intended performance envelope.

Our tests are aimed at the first three of these items.

The results of our experiments show that 91% of all attempts to launch an application, write a file other than those sanctioned or to open an un-sanctioned port or to inappropriately modify a MAF data structure were detected and correctly diagnosed. Finally we note that there are no false positives. This is to be expected if the System Architectural Model is a reasonable abstraction of the program.

4 Conclusions

PMOP and AWDRAT provide valuable services; but at what cost? There are two major costs involved in the use of these systems: The development cost of building a System Architectural Model and the runtime overhead imposed by monitoring the application. In our experience so far, the second of these costs is negligible, particularly since the target system in our experiments was an interactive system.

The development cost of building the System Architectural Model depends on how well one understands the target system's architecture and how hard it is to translate that understanding into the formalism used for our System Architectural Model.

In the best of cases, the application to be protected is well understood and reasonably well modularized. However, this isn't normally the case for legacy applications; they are typically poorly documented. Furthermore, the documentation that exists is usually out of sync with the actual system code. All these were true of our target system; however, the availability of our wrapper technology made it considerably easier for us to to engage in the necessary "software archeology" to gain an understanding of the system before constructing its System Architectural Model.

Once the architecture of the application was understood, the construction of the System Architectural Model was conceptually straightforward; however, the actual coding of the System Architectural Model in our current plan language is rather tedious and might well be more easily generated from a graphical language (e.g. UML). The core of the MAF system itself is on the order of 30,000 lines of Java code while The System Architectural Model that we built to describe it is 448 lines of code, together with other ancillary code that provided greater services, our coding effort amounted to about 8% of the size of the target system.

AWDRAT and PMOP are frameworks to which an application system may be tethered in order to provide survivability properties such as error detection, fault diagnosis, backup and recovery. They remove the concern for these properties from the domain of the application design team, instead providing these properties as infrastructure services. They use cognitive techniques to provide the target system with the self-awareness and self-adaptivity necessary to monitor its behavior, diagnose failures, adapt and recover from both insider and external attackers. This frees application designers to concentrate on functionality instead of exception handling, and provides a framework for ensuring a high level of system survivablility, independent of the skills of the application designers.

References

[1] R. Balzer and N. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *Proceedings of the First Darpa Information Security Conference and Exhibition (DISCEX-II)*, volume II, pages 361–368, Jan 25-27 2000.

[2] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, , and D. Moon. Common lisp object system specification. Technical Report 88-002R, X3J13, June 1988.

[3] J. deKleer and B. Williams. Diagnosis with behavior modes. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989.

[4] T. Hollebeek and R. Waltzman. The role of suspicion in model-based intrusion detection. In *Proceedings of the 2004 workshop on New security paradigms*, 2004.

[5] S. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Number ISBN 0-201-17589-4. Addison-Wesley, 1989.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[7] R. Laddaga, P. Robertson, and H. E. Shrobe. Probabilistic dispatch, dynamic domain architecture, and self-adaptive software. In R. Laddaga, P. Robertson, and H. Shrobe, editors, *Self-Adaptive Software*, pages 227–237. Springer-Verlag, 2001.

[8] C. Rich. Inspection methods in programming. Technical Report AI Lab Technical Report 604, MIT Artificial Intelligence Laboratory, 1981.

[9] C. Rich and H. E. Shrobe. Initial report on a lisp programmer's apprentice. Technical Report Technical Report 354, MIT Artificial Intelligence Laboratory, December 1976.

[10] H. Shrobe. Dependency directed reasoning for complex program understanding. Technical Report AI Lab Technical Report 503, MIT Artificial Intelligence Laboratory, April 1979.

[11] H. Shrobe. Model-based diagnosis for information survivability. In R. Laddaga, P. Robertson, and H. Shrobe, editors, *Self-Adaptive Software*. Springer-Verlag, 2001.

[12] H. Shrobe. Computational vulnerability analysis for information survivability. In *Innovative Applications of Artificial Intelligence*. AAAI, July 2002.

[13] H. Shrobe, R. Laddaga, B. Balzer, N. Golman, D. Wile, M. Tallis, T. Hollebeek, and A. Egyed. Awdrat: Awdrat: A cognitive middleware system for information survivability. In *Innovative Applications of Artificial Intelligence*. AAAI, July 2006.