

AWDRAT: A Cognitive Middleware System for Information Survivability

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

- The infrastructure of modern society is controlled by software systems that are vulnerable to attacks. Many such attacks, launched by "recreational hackers" have already led to severe disruptions and significant cost. It, therefore, is critical that we find ways to protect such systems and to enable them to continue functioning even after a successful attack.

This article describes AWD RAT, a prototype middleware system for providing survivability to both new and legacy applications. AWD RAT stands for architectural differencing, wrappers, diagnosis, recovery, adaptive software, and trust modeling. AWD RAT uses these techniques to gain visibility into the execution of an application system and to compare the application's actual behavior to that which is expected. In the case of a deviation, AWD RAT conducts a diagnosis that determines which computational resources are likely to have been compromised and then adds these assessments to its trust model. The trust model in turn guides the recovery process, particularly by guiding the system in its choice among functionally equivalent methods and resources. AWD RAT has been applied to and evaluated on an example application system, a graphical editor for constructing mission plans. We describe a series of experiments that were performed to test the effectiveness of AWD RAT in recognizing and recovering from simulated attacks, and we present data showing the effectiveness of AWD RAT in detecting a variety of compromises to the application system (approximately 90 percent of all simulated attacks are detected, diagnosed, and corrected). We also summarize some lessons learned from the AWD RAT experiments and suggest approaches for comprehensive application protection methods and techniques.

Overview

To the extent that traditional systems provide for immunity against attack, they do so using one of two approaches: The first approach uses a library of known, suspected, or hypothesized patterns of attack and attempts to match the observed behavior of the system against patterns in the library. The second approach constructs a statistical model of "typical" behavior then detects statistically anomalous behavior that deviates from the typical profile. Neither of these approaches is satisfactory: The first approach fails in the face of novel attacks, producing an unacceptably high false negative rate; in practice, the advantage goes to the attacker since the attacker can produce novel attacks more rapidly than defenders can respond to them. The second approach confounds unusual behavior with illegal behavior; this produces unacceptably high false positive rates and lacks diagnostic resolution even when an intrusion is correctly flagged (Allen et al. 2000; Debar, Dacier, and Wespi 1999; Lunt 1993; Axelsson 1998).

In this article, we present a different approach embodied in AWD RAT, a middleware system to which an existing application software (the "target" system) may be retrofitted. AWD RAT provides immunity to compromises of the target system by making it appear to be self-aware and capable of actively checking that its behavior corresponds to that intended by its designers. "AWDRAT" stands for architectural differencing, wrappers, diagnosis, re-

covery, adaptivity, and trust modeling. We will explain each of these facilities in detail later on.

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.

Intended Range of Application

AWDRAT is intended to be a general-purpose utility that can be applied to a broad class of target systems. We have experimented with the application of AWDRAT to both server and interactive application system code.¹ In general, AWDRAT is intended to be useful for mission-critical applications that might be subjected to attack by malicious outsiders. However, AWDRAT does impose some overhead in both performance and space consumption; the most appropriate targets for its use, therefore, are large, mission-critical applications that do not need to meet hard real-time performance demands but that are required to provide useful services on a continuous basis.

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, that is, an existing body of code (which we'll refer to as the "target 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 (for example, 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.

In contrast to traditional protection mechanisms mentioned earlier, which focus on intrusion detection, AWDRAT's primary concern is with the detection of misbehavior. AWDRAT is only secondarily concerned with the vector by which the attacker gained access to the target system. Intrusion detection systems have the advantage of being proactive; when they work, they prevent the attacker from corrupting the target system. AWDRAT, on the other hand, tries to repair the target system once a misbehavior has been detected. The two approaches should therefore be seen as complementary that might both be parts of an overall "defense in depth" strategy.

Roadmap

The remainder of this article is organized as follows. First we describe the overall architecture of AWDRAT, showing both its major components and internal models. We then go on to describe each of the major components of AWDRAT. First we describe how AWDRAT synthesizes wrappers that instrument the target system and gather data about the target system's execution. Next we describe "architectural differencing," the process of comparing the actual behavior of the system (gathered by the instrumentation we will describe) to the behavior predicted by the system architectural model. In the next section we describe how AWDRAT's diagnostic system uses a description of the deviation between predicted and observed behavior to produce a diagnosis. The following section describes how the AWDRAT framework for adaptive software allows the application to provide multiple methods for achieving its goals while the last of these sections describes how such adaptive behavior can be used to avoid using compromised resources during and after recovery.

In the succeeding section, we describe a series of experiments we performed in applying AWDRAT to a particular target system. This section details the types of attacks that were simulated and shows how well AWDRAT performed in detecting, diagnosing and recovering from these simulated attacks. Finally, the last section discusses what additional steps are needed in order to begin deploying AWDRAT on real target systems.

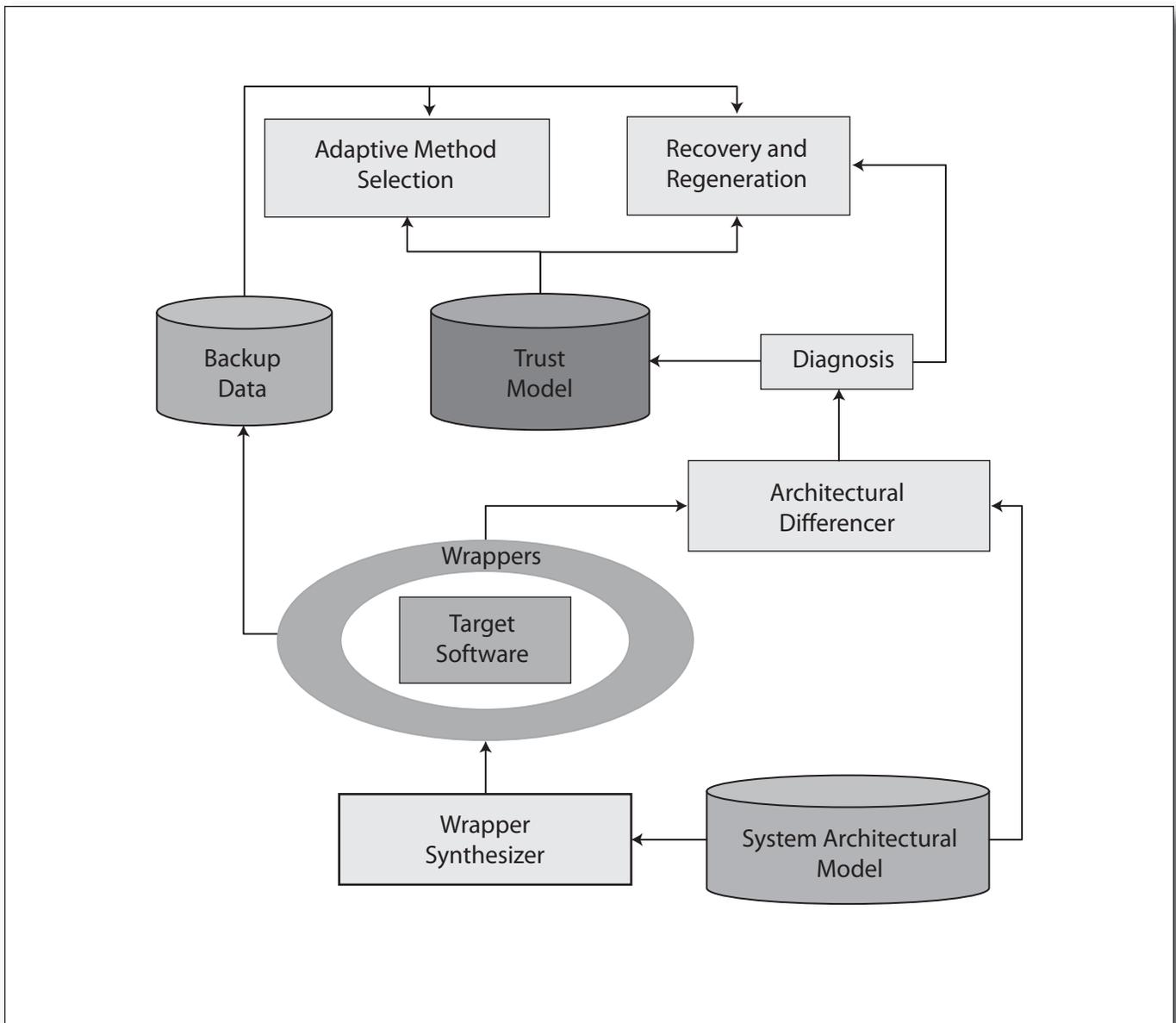


Figure 1. The AWD RAT Architecture.

The AWD RAT Architecture

The AWD RAT architecture is shown in figure 1. This architecture includes both a variety of models maintained by AWD RAT (the round boxes in the figure) as well as a number of major computational components (the square boxes in the figure). AWD RAT is provided with a model of the intended behavior of the target system being protected by AWD RAT (the system architectural model in the figure). This model is based on a “plan level” decomposition that provides pre- and post- and invariant conditions for each module of the target system. AWD RAT actively enforces these declarative models of intended behavior using “wrap-

per” technology. The Wrapper Synthesis module in the figure is responsible for synthesizing nonbypassable 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 checks 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 sent to the AWD RAT component labeled

Diagnosis in the figure. The AWD RAT 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. AWD RAT proceeds to use the results of the diagnosis to calculate the types of compromise that may have affected each computational resource of the target system. AWD RAT also calculates the likelihood of each possible compromise. These results are stored in an internal model, labeled Trust Model in the figure.

Next, AWD RAT 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). During and after recovery, AWD RAT tries to help the target avoid using compromised resources (which would cause the target system to fail again). In some cases, the target system may have more than one method for accomplishing a task; in others, the AWD RAT infrastructure could provide alternative methods. For example, the target system might use a library method for reading image files that is very fast, but prone to buffer overflow attacks. However, there is an alternative library that is slower but immune to these attacks.² The Alternative Method Selection module 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 (for example, 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 AWD RAT. 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 (for example, because they run faster, or render better images). The Adaptive Method Selection module, therefore, attempts to find a combination of method and resources that makes a good trade-off, maximizing the quality of service rendered and minimizing the risk of system failure.

AWD RAT also uses the target system's system architectural model to recognize the critical data that must be preserved in case of failure. AWD RAT'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, AWD RAT uses these backup copies to repair compromised data resources; in addition, the AWD RAT Adaptive Method Selection module may decide to use the backup copies of data instead of the primary copy.

Using this combination of technologies, AWD RAT provides "cognitive immunity" to both intentional and accidental compromises. An application that runs within the AWD RAT 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. AWD RAT 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.

Synthesis of Wrappers and Execution Monitor

One of the key elements of AWD RAT is its use of wrapper technologies. The AWD RAT architecture is very general purpose 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 or operating system environment. Our current set of wrapper technologies limits us to Java and C programs running in a Windows environment. Other than this, we make very few assumptions about the nature of the target program; for example, it may be single or multithreaded, it can be a server or an interactive application program.

AWD RAT employs two distinct wrapper technologies: SafeFamily (Balzer and Goldman 2000; Hollebeek and Waltzman 2004) and JavaWrap. The first of these encapsulates system DLL's, allowing AWD RAT to monitor any access to external resources such as files or communication ports. The second of these provides method wrappers for Java programs, providing a capability similar to "around" methods in the Common-Lisp Object System (Keene 1989; Bobrow et al. 1988) or in Aspect-J (Kiczales et al. 2001). 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

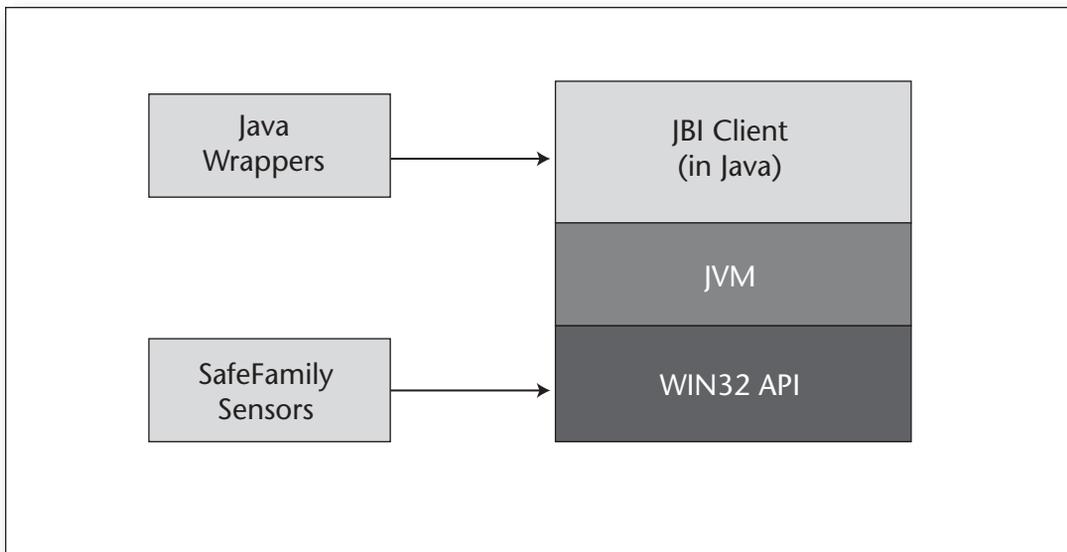


Figure 2. Two Types of Wrappers Used in AWD RAT.

to invoke the original method if desired. To use the SafeFamily facility, one must provide an XML file of rules specifying the resources (for example, files, ports) and actions (for example, writing the file, communicating over the port) that are to be prevented. 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 run-time model. Together they provide AWD RAT with the ability to monitor the applications behavior in detail as is shown in figure 2.

The inputs to these two wrapper-generator facilities (the JavaWrap XML spec, the Java Mediator files, and the SafeFamily XML specification file) are not provided by the user but are instead automatically generated by AWD RAT from a “system architectural model” such as that shown in figure 3. The system architectural model is written in a language similar to the “Plan Calculus” of the Programmer’s Apprentice (Rich and Shrobe 1976; Shrobe 1979; Rich 1981); 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 postconditions. In AWD RAT, 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 (for example, write to a file). Each component of the system architectural model may be annotated with “entry events,” “exit events,” “allowable events,” and “prohibited

events.” Entry and exit events are described by method specifications (and are caught through the JavaWrap facility); allowable and prohibited events may be either method calls or resource access events (resource access events are caught by the SafeFamily facility). The occurrence of an entry (exit) event indicates that a method that corresponds to the beginning of a component in the system architectural model has started (completed) execution. Occurrence of a prohibited event is taken to mean that the application system has deviated from the specification of the model.

Given this information, the AWD RAT 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 as is shown in figure 4.

Architectural Differencing

In addition to synthesizing wrappers, the AWD RAT generator also synthesizes an “execution monitor” corresponding to the system architectural model as shown in figure 4. 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 5. Should a deviation be detected, diagnosis and recovery is attempted. Our diagnosis and recovery systems, far and away the most complex parts of the AWD RAT run-time system, are

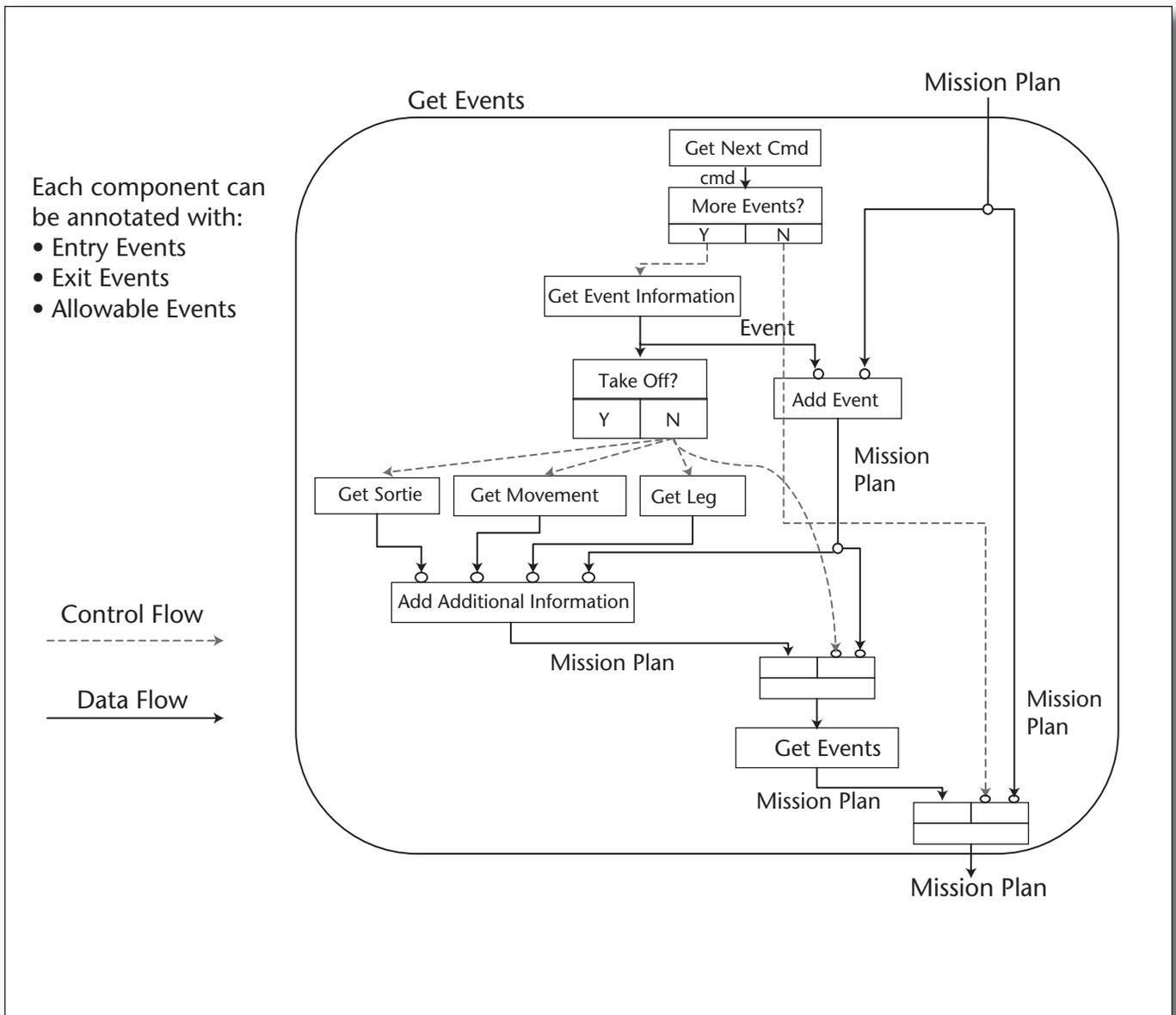


Figure 3. An Example System Architectural Model.

written in Common Lisp; therefore, the actual “plumbing” generated consists of Java wrappers that are merely stubs invoking Lisp mediators that, in turn, signal events to the execution monitor, which is also written in Lisp. This is shown in figure 6.

The system architectural model provided to AWD RAT includes prerequisite and postconditions for each of its components. A special subset of the predicates used to describe these conditions is built into AWD RAT and provides a simple abstract representation of data structuring. The AWD RAT synthesizer analyzes these statements and generates code in the Lisp mediators that creates backup copies of those da-

ta structures that are manipulated by the application and that the system architectural model indicates are crucial.

The execution monitor behaves as follows: Initially all components of the system architectural model are inactive. When the application system starts up it creates a “startup” event for the top-level component of the model, and this component is put into its “running” state. When a module enters the “running” state it instantiates its subnetwork (if it has one) and propagates input data along data flow links and passes control along control flow links.

When data arrives at the input port of a component, the execution monitor checks to

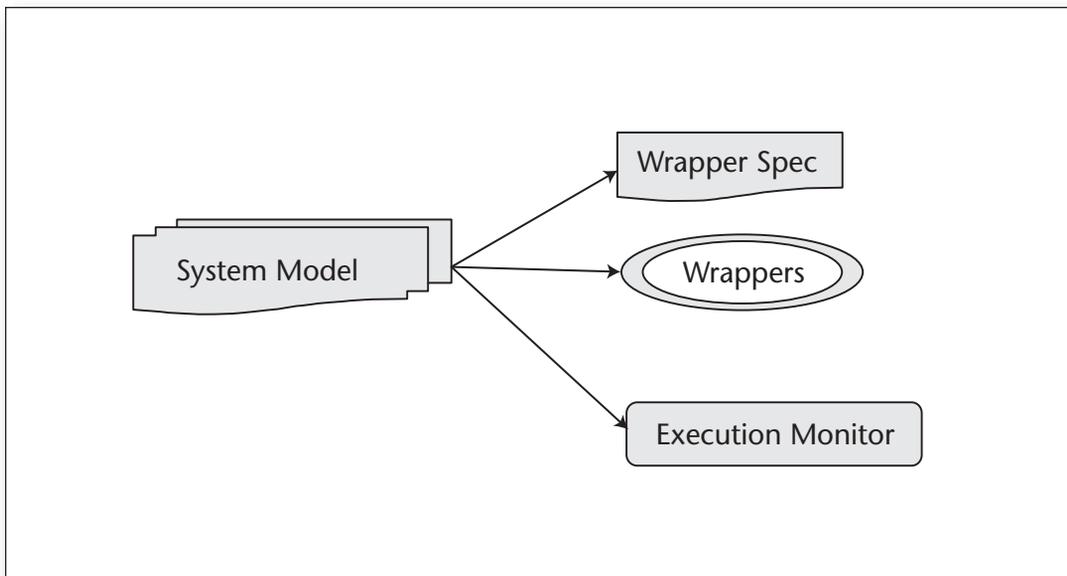


Figure 4. Generating the Wrapper Plumbing.

see if all the required data is now available; if so, the execution monitor checks the preconditions of this component, and if they succeed, it marks the component as “ready.” Should these checks fail, diagnosis is initiated.

As events arrive from the wrappers, each is checked. (1) If the event is a “method entry” event, then the execution monitor checks to see if this event is the initiating event of a component in the “ready” state; if so, the component’s state is changed to “running.” Data in the event is captured and applied to the input ports of the component. (2) If the event is a “method exit,” then the execution monitor checks to see if this is the terminating event of a “running” module; if so, it changes the state of the component to “completed.” Data in the event is captured and applied to the output ports of the component. The component’s postconditions are checked and diagnosis is invoked if the check fails. (3) Otherwise the event is checked to see if it is an allowable or prohibited event of some running component; detection of an explicitly prohibited event initiates diagnosis as does the detection of an unexpected event, that is, one that is neither an initiating event of a ready component or a terminating or allowable event of a running component.

Using these generated capabilities, AWD RAT detects any deviation of the application from the abstract behavior specified in its system architectural model and invokes its diagnostic services.

Diagnostic Reasoning

AWDRAT’s diagnostic service is described in more detail in Shrobe (2001) and draws heavily on ideas in deKleer and Williams (1989). Each component in the system architectural model provided to AWD RAT 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 the section on Architectural Differencing, 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 database 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, postconditions of any component are justified as depending on the assumption that the component has executed normally as is shown in figure 7. This is similar to the reasoning techniques in Shrobe (1979).

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

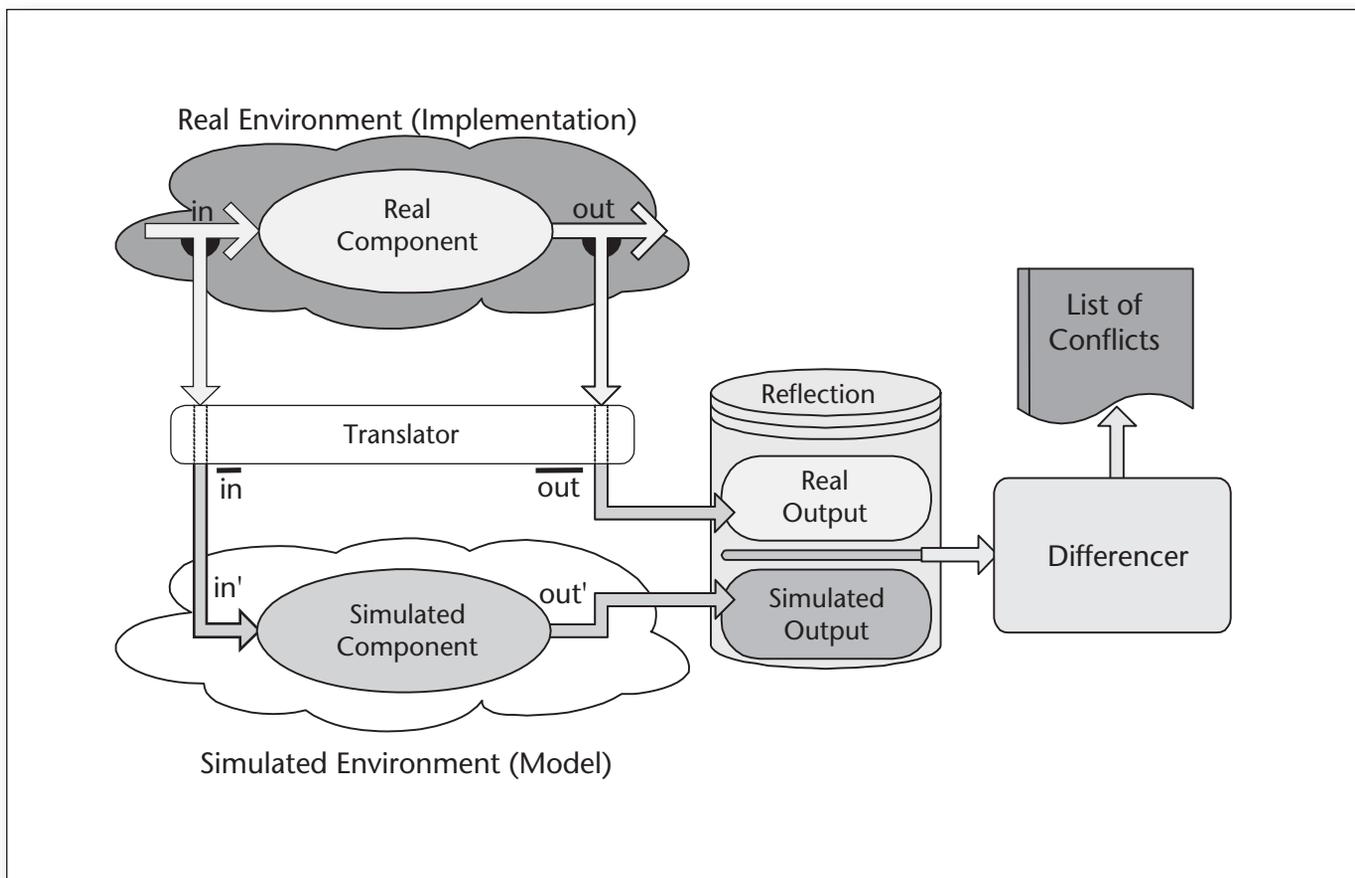


Figure 5. Architectural Differencing.

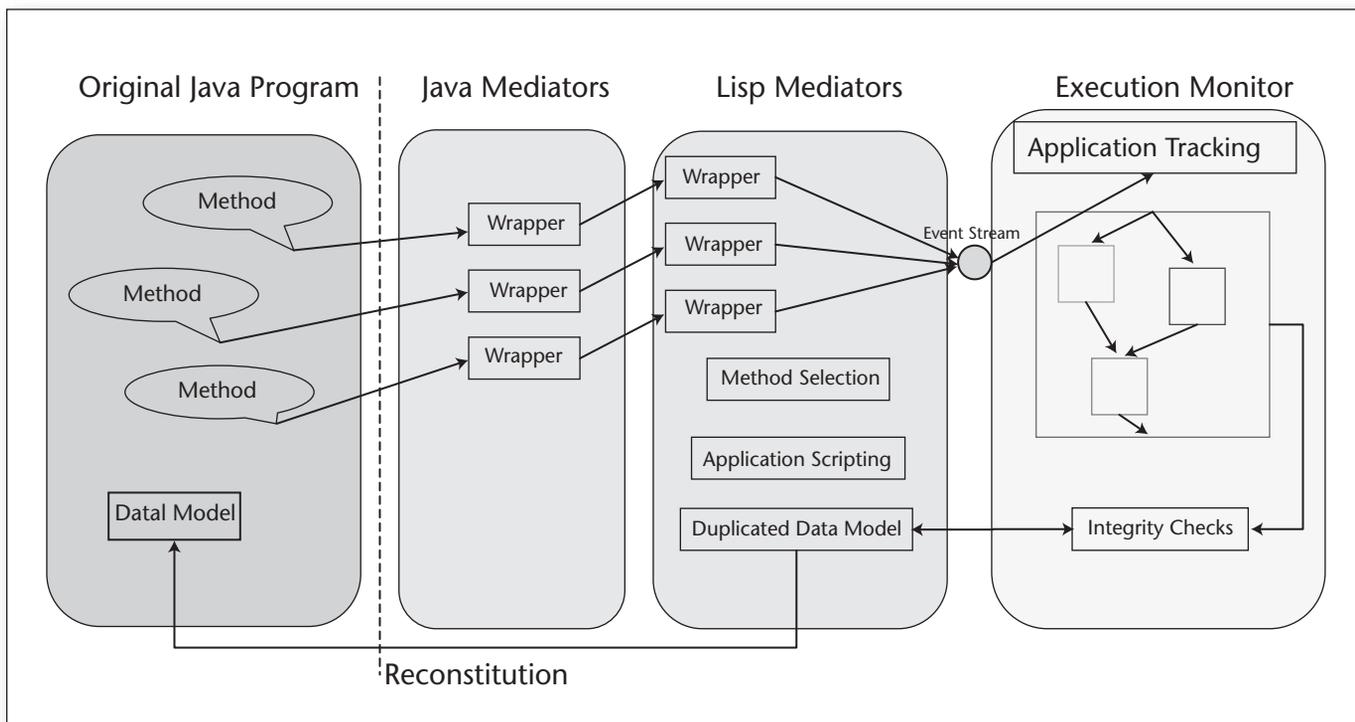


Figure 6. The Generated Plumbing.

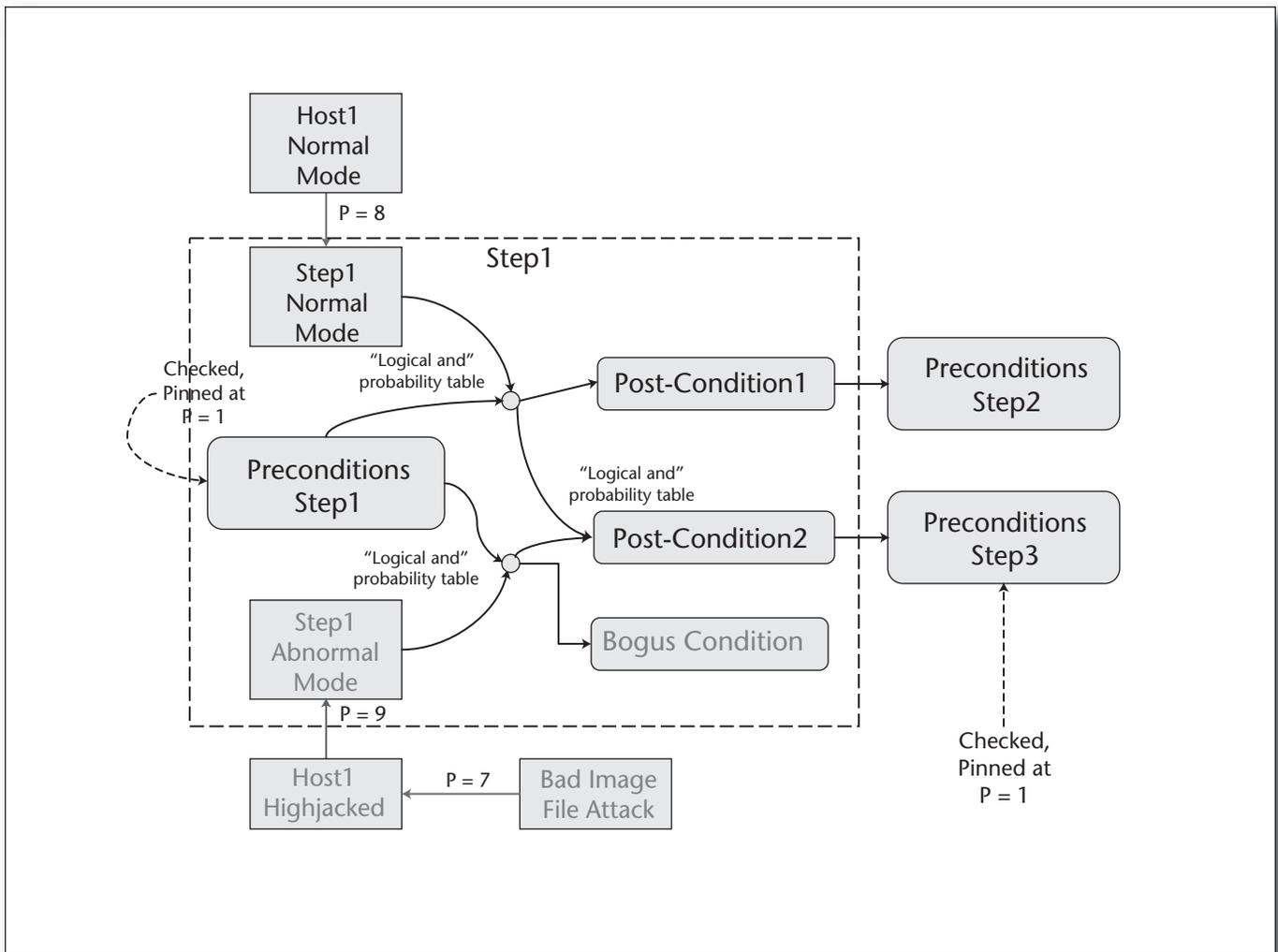


Figure 7. Dependency Graph.

these specifications of off-nominal behavior are substituted.

In addition to modeling the behavior of the components in the system architectural model, AWD RAT 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 (for example, class files), and even the binary representation of the code in memory. Part of the system architectural model provided to AWD RAT 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, AWD RAT'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 8. AWD RAT'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.

Self-Adaptive Software

Recovery in AWD RAT depends critically on self-adaptive techniques such as those described in Laddaga, Robertson, and Shrobe (2001). The critical idea is that in many cases an application may have more than one way to perform a task. For example, in the experiments that will be described in the section on experimental methods, we tethered a graphical editor application to AWD RAT. This application loads image files (for example, gif, jpeg) and, as it happens, there is a vulnerability

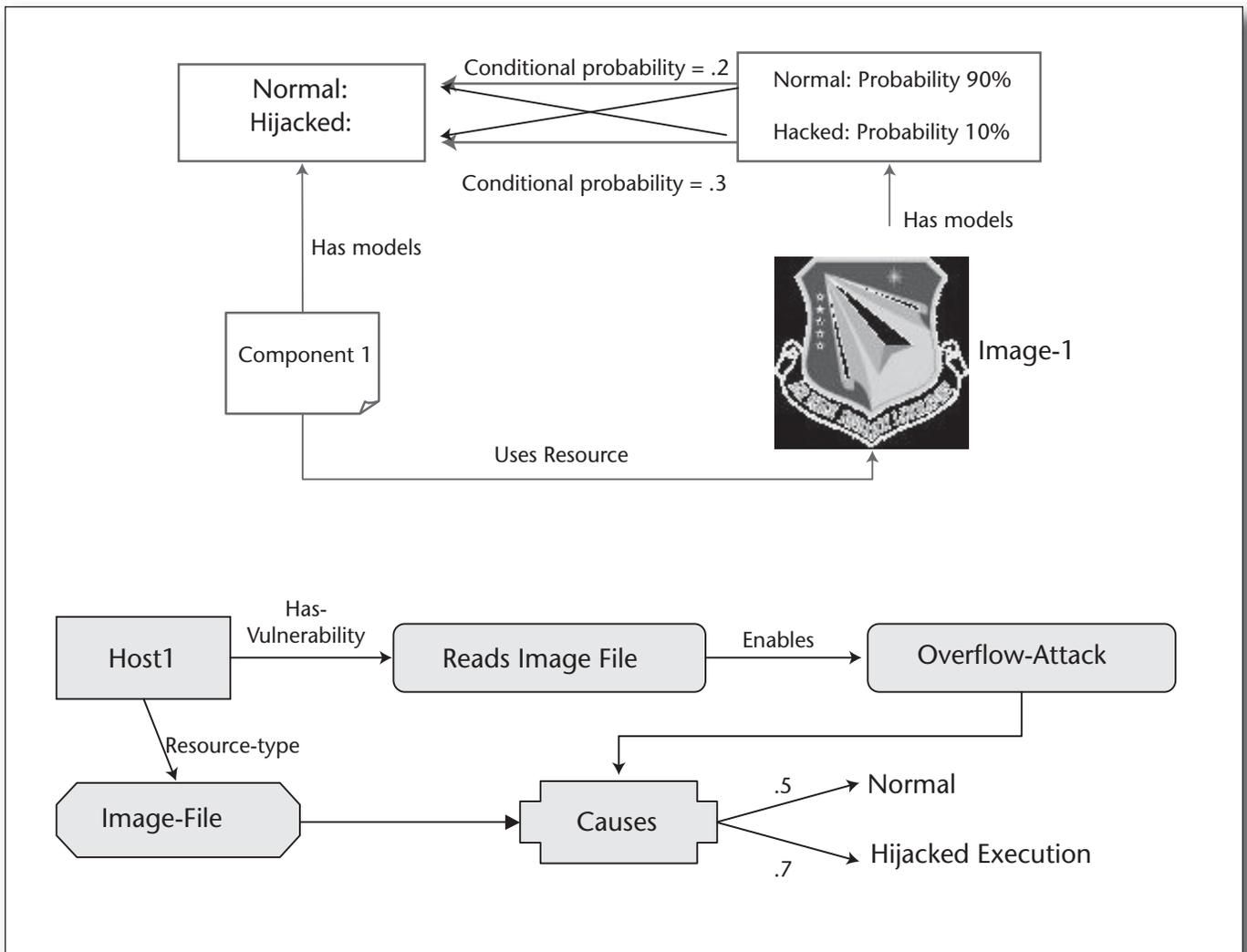


Figure 8. Diagnosis with Fault and Attack Models.

(since fixed) related to loading malformed image files. This is enabled by the use of a “native library” (that is, code written in C). There is also a Pure Java library that performs the same task, however, it is slower, handles fewer image formats, and also produces lower-quality images in some cases.

Self-adaptive software involves making dynamic choices between alternative methods such as the native and Pure Java image-loading 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 “nonfunctional 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 9, the task of AWD RAT’s adaptive software facility is to pick that method and set of resources that will deliver the highest net benefit. Thus AWD RAT’s self-adaptive software service provides a decision theoretic framework for choosing between alternative methods.

Recovery and Trust Modeling

As shown in figure 10, 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 as-

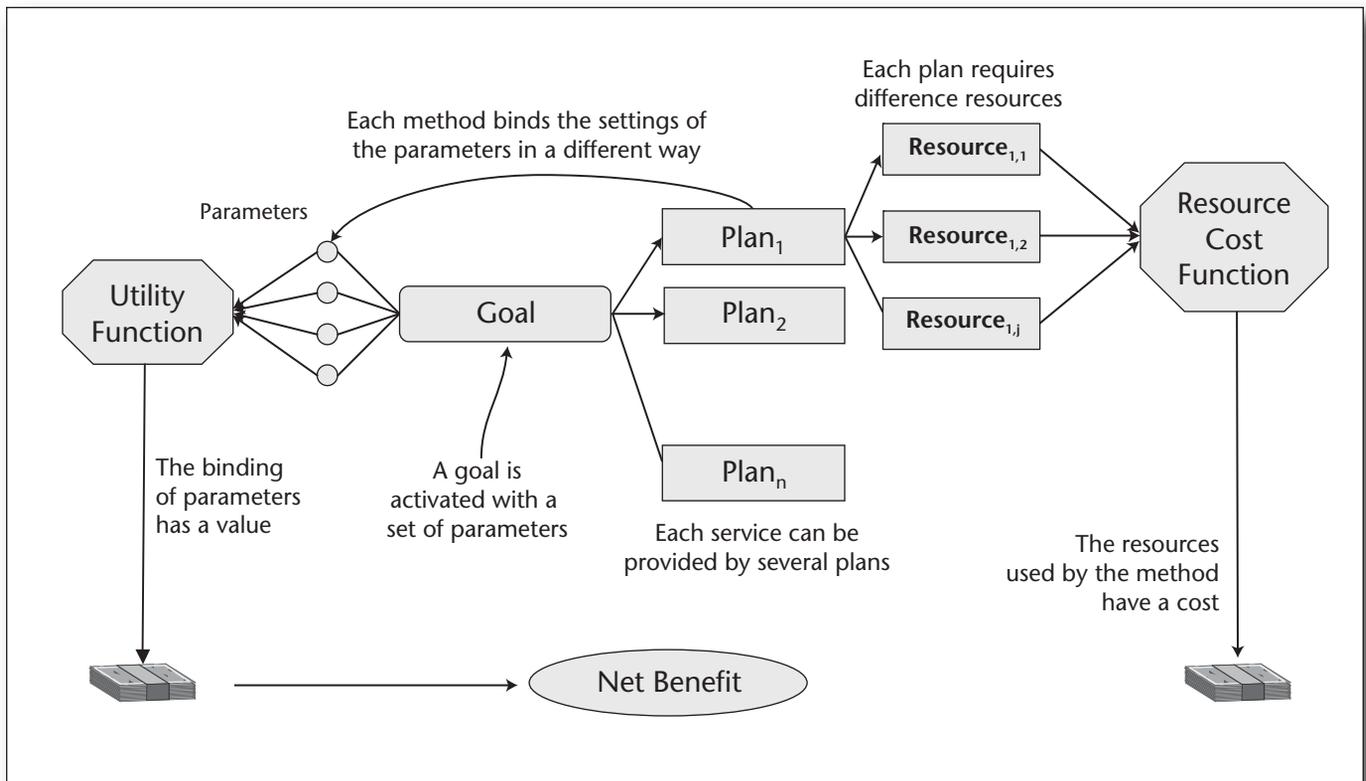


Figure 9. Adaptive Software Picks the Best Method.

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 (1) restarting the application or dynamically reloading its code files (assuming that the application system's language and runtime environment support 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; (2) 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; (3) 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 (for example, 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 (that is, 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.³ 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.

Experimentation and Results

AWDRAT's goal is to guarantee that the target system tethered to it faithfully executes the intent of the software designer; for example, for

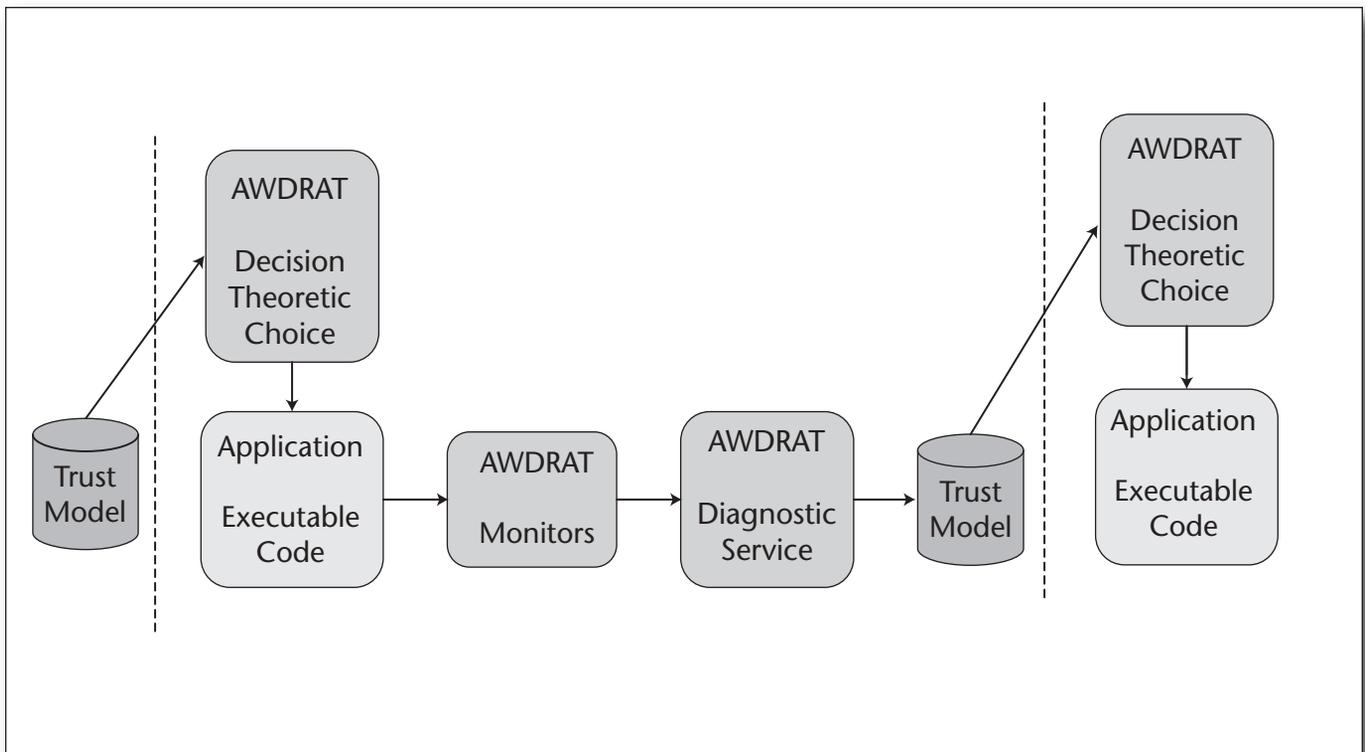


Figure 10. The Trust Model Guides Future Recovery.

an interactive system this means that the system should faithfully execute the commands specified by its user (for example, through an application GUI), or for a server application this means that it should faithfully execute the requests received from its client applications.

To assess AWD RAT we applied it to the defense of a particular application system, the MAF interactive mission planner—a component of the DARPA Demval demonstration system, which is in turn based on Rome Labs Joint Battlesphere Infrastructure (JBI). MAF is an interactive graphical editor for mission plans. Its basic structure is one similar to many such systems: User commands are entered as “button clicks” and in response to each such click the system invokes a method that handles the command by updating both the internal data structures of the program and the corresponding graphical display. It largely follows the paradigm of a “model-view-controller” system. MAF also makes requests of the JBI core’s publish-and-subscribe server; however, the MAF system itself is not a server that responds to client program’s network requests. MAF is implemented in JAVA.

Due to its structure and implementation language, the MAF system lacks many vulnerabilities common to server systems written in unsafe languages such as C. The primary

vulnerabilities in MAF arise either through use of native libraries containing unsafe code or through an attack that somehow manages to gain enough privileges to modify the application’s class files. AWD RAT’s primary concern is not with the attack vector, but rather with the identification and containment of misbehavior and with the identification of corrupted resources so that the recovery processes can avoid them. In particular, AWD RAT focuses on the maintenance of data integrity and data privacy.

We identified four properties that AWD RAT should guarantee. First, the data structures maintained by the application should accurately reflect user requests—each field of each data structure should be what the GUI requested, no other fields should be present, and all the fields requested by the GUI should be present. Second, the application should not be able to open a port to any application other than to those JBI servers that it must communicate with (that is, the JBOSS server). Third, the application should not be able to open a port to a foreign host. Finally, the application should not be able to write files except in the specified set of directories that constitute its temporary storage areas.

Although these are stated in terms of the MAF system in the context of the JBI environ-

ment, 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.

Experimental Methodology

To test the effectiveness of AWD RAT in guaranteeing these properties we adopted three distinct ways of attempting to violate the properties: modification of class files, random attacks, and wrapped methods.

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 AWD RAT 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 (for example, 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. AWD RAT 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. The core AWD RAT machinery is kept unaware of the attack machinery. Its goal is to detect and characterize a violation.

The MAF's data structures are a relatively simple tree, built from instances of a few classes. Each flight path (mission) is represented by a "Mission-Builder" that contains a set of "Events," "Legs," "Sorties," and "Movements." An Event is a "Take-off," a "Landing," or a

"Waypoint." For each "Take-Off" event there is a corresponding "Leg," "Sortie," and "movement." The top-level data structure is an instance of the Mission-Builder class, containing a hash table for the overall mission data and four additional hash tables holding the sets of event, leg, sortie and movement data structures. Each entry in these tables is an instance of the appropriate type, containing a hash table with the data specific to that element of the mission plan.

When considering data-structure integrity, it is important to understand that the data structures can be modified using two different levels of calls. Each data structure is implemented as a Java class with accessor methods (for example, "setInformation," "getInformation"). In addition, Java defines methods on hash tables (for example, "put," "get"). The application always accesses the data structures through its API using the "getInformation" and "setInformation" methods. However, attack code might access the data structures below this level of API (for example, using the hash table get and put methods or even using lower-level memory-accessing capabilities at the native code level). Thus it is necessary to simulate attempts to corrupt the data structures using both the API methods and the hash table methods (AWD RAT does not wrap or monitor the hash table-level methods for both pragmatic and technical reasons, so modifications at this level are as opaque as modifications using native code-level capabilities).

Each of these simulated attack methods has drawbacks and strengths: The first method allows very precise placement of an attack within the code, but might be criticized for being overly synchronized to the normal execution flow of the program; the random attacks have a dual set of issues. However, our feeling is that collectively, these tests cover a great deal of the appropriate space. Although we can't be particularly rigorous about this, we observe that for an attack to be effective it must (1) cause the target system to 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; and (5) fail to meet its intended performance envelope.

Our tests are aimed at the first three of these items. Figure 11 shows the tests, conducted by category, in more detail.

The use of a wrapper to simulate an attack payload was only employed for the loadImage method (to avoid actually building a payload carrying corrupted image file). The wrapper on-

<p>1. Modification of class files: Insert call to write files Insert call to launch external application Insert call to open port Insert code to modify the data structures at the API level Insert code to modify the data structures below the API level</p> <p>2. Random delay attacks (through a parallel thread) Make call to write files Make call to launch external application Make call to open port Attempt to modify the data structures at the API level Attempt to modify the data structures below the API level</p> <p>3. Wrapped methods (used only for “loadImage”) Make call to write files Make call to launch external application Make call to open port</p>
--

Figure 11. Tests Conducted by Category.

ly attempts to do the disallowed action when using the native loadImage library on specific image files identified in a configuration file; this is admittedly a bit hokey since the same wrapper mechanism is used both to launch the simulated attack and to make the decision as to whether to use the native or pure Java load method, but we have tried to keep the two separate and to run an honest test. In this case we do not attempt to corrupt the MAF data structures because loadImage is only called in the application during its initialization, which is before the data structures are created, so image-based attacks only attempt to open a port or to write a file.

The second category of violation is launched from a thread that is started by the initialization code of the system. This thread waits until the user begins to enter a mission plan, then picks an arbitrary delay time (less than 4 minutes); after that delay time, it either attempts to open a port, write a file, or to corrupt the data structures. To do the last of these, it picks an arbitrary element of the MAF data structures and attempts to either modify an existing field of the data structure or to add a new field. Strictly speaking, adding a new field to the data structures is harmless, the application will ignore the extra field. However, the criterion for success is detecting any deviation of the application from the actions requested by the GUI, so we include these case as well.

Detection Methods

As explained in the sections on wrappers, architectural differencing, and diagnostic reasoning, AWD RAT picks up violations in one of three ways: (1) It checks the integrity of the Java data structures against its internal backup copy everywhere that the system architectural model specifies that the data structures should be consistent. (2) It checks that monitored methods are called only at points in the execution sanctioned by the system architectural model. (3) It receives messages from the SafeFamily (DLL) wrappers, alerting it to violations of the access rules imposed by SafeFamily. Some violations that are conceptually in the same category (for example, data structure integrity) are picked up by more than one mechanism. For example, an attempt to modify the MAF data structures using an API-level call is usually picked up because the call isn't sanctioned at that point of the execution; however, using a hash table method on a hash table held in one of the data structures will be picked up by the integrity check, since the hash table methods aren't wrapped.

Results Summary

The results of our experiments are shown in table 1. These show that all attempts to launch an application, write a file other than those sanctioned, or open an unsanctioned port were detected. The only exception to this broad statement is that an attacker can write to a file in the MAF's temporary directory or open one of the ports used by AWD RAT itself (however, opening such a port would violate the rules of engagement for the experiments). Almost all attempts to destroy the integrity of the MAF data structures were detected; the exception is when the modification is made using the MAF API-level calls during the execution of a method that legitimately uses the same API call. This occurs only in hand-modified source code “attacks” (one “random” attack managed to tickle this case). In principle, it's possible that an attack operating below the MAF API level could modify the MAF data structures and that the modification could be overwritten later by the uncorrupted MAF code doing the right thing. For example:

The GUI requests that the “ALT” field of Event 1 be set to “30000”

The attack code in another thread sets the “ALT” field of Event 1 to “1” using hash table or lower-level calls

The MAF method sets the “ALT” field of Event 1 to “30000”

The net effect is that the data structures are uncorrupted; however, AWD RAT's machinery

Attack Type	Number of Attempts	Detected	Diagnosed	Corrected
Hand Placed	24	20/83%	20/83%	11/73%
Data MAF API	10	6/60%	6/60%	6/60%
Data lower API	5	5/100%	5/100%	5/100%
Port Opening	3	3/100%	3/100%	NA
File Write	3	3/100%	3/100%	NA
Application Launch	3	3/100%	3/100%	NA
Random	26	25/96%	25/26%	13/92%
Data MAF API	6	5/83%	5/83%	5/83%
Data lower API	8	8/100%	8/100%	8/100%
Port Opening	5	5/100%	5/100%	NA
File Write	4	4/100%	4/100%	NA
Application Launch	3	3/100%	3/100%	NA
Wrapped	9	9/100%	9/100%	9/100%
File Write	3	3/100%	3/100%	3/100%
Port Opening	3	3/100%	3/100%	3/100%
Application Launch	3	3/100%	3/100%	3/100%
Total	59	54/91%	54/91%	33/86%

Table 1. Experimental Results

will never detect the unsuccessful attempt to corrupt the data structures in this case.

Discussion. The first category of attack includes hand-inserted attack code. Four of these included calls to MAF API-level methods inside other routines that legitimately invoke these methods. These attacks could have been caught if AWD RAT had interposed itself at a higher level in the call tree, specifically at the boundary between the GUI and the rest of the MAF application. As it happens, the MAF system is not appropriately modularized to facilitate this. One of the random attacks managed by pure chance to run in parallel with an application routine that invoked the same MAF-level API; we thought this was too unlikely to actually occur, but we were wrong. This slips by because the current AWD RAT execution monitor isn't aware of thread structure.

The lesson to be learned is that for AWD RAT to be effective, the abstract modularity of the

target system should be reflected in its actual implementation. In particular, since wrapping takes place at the level of the individual method, it is important that each method in the code (that is, Java method for the MAF system) should correspond to no more than one component of the system architectural model (however, a component of the system architectural model can be implemented by a set of methods in the code).

Attacks that attempt to open ports, write files, or launch applications were intercepted and blocked by the SafeFamily wrappers, preventing any bad effect from being propagated. This is why the last column is marked Not Applicable (NA) for these categories of attack. In fact, AWD RAT does restart the application and rebuild its data structures in these cases as well. For the Wrapped cases (that is, those involving simulated corrupt image files) the last column is listed because the dominant diagnostic hypothesis in those cases is that an attack was

launched from payload code embedded in the image being loaded. In these cases, switching to the Pure Java method or using a different format of the image file constitutes successful recovery. We did not mark these cases as NA, since there was significant decision making in the recovery process. In the other cases, the dominant diagnostic hypothesis was that the class files (or core image) were corrupted, in which case the recovery process involved switching the class path to backup copies of the JAR files.

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.

In addition to these extensive internal tests, we also subjected AWDROT to a Red-Team experiment. The Red-Team experimented with a much broader range of issues than the internal experiments, many of which involved issues that AWDROT was not expected to deal with. Nevertheless AWDROT performed at a level above the programmatic goals of the DARPA SRS program that sponsored this effort.

Moving Toward Practicality

AWDROT is still a prototype system and it has been applied to a single major application system; it is not yet ready for full deployment. It is worth pulling back at this point to assess what we have learned and to speculate on how the AWDROT framework might develop over time. We will address the following issues in turn: (1) what is the cost of applying AWDROT? (2) where does AWDROT fail to provide general-purpose solutions? (3) what is the long term vision for self-protective and self-regenerative systems?

The Costs of Applying AWDROT

AWDROT provides protection and regenerative services; but at what cost? Would it be easier to engineer these services into each application by hand, or is there an advantage to providing a general framework for self-protection and regeneration? Is the overhead imposed by the general framework acceptable?

There are two major costs involved in the use of AWDROT: (1) the development cost of building a system architectural model and (2) the run-time overhead that AWDROT imposes on the hosted application system. In our experience so far, the second of these costs is negligible. Since we used AWDROT to defend an interactive application, the key question is whether the monitoring and checking overhead slows down the system sufficiently to impair the user's experience. In the experiment re-

ported on here, we experienced no observable degradation of user interface performance. However, for real-time and embedded applications, the question is certainly still open. At the least, we will need to be much more judicious with the use of monitoring wrappers and we will similarly have to be more cognizant of the need to contain the cost of checking that the application is behaving consistently with its system architectural model.

The development cost of building the system architectural model depends on two things: How well understood is the system's architecture? How easy is it to build a system architectural model for a well understood architecture?

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. In our experiment with the MAF system, we were working with prototype, demonstration code whose architecture was not documented; we had to engage in "software archeology" to gain an understanding of the system before constructing its system architectural model. One of the key tools we used for this effort was AWDROT's wrappers, which allowed us to trace the execution of the application system at the method call-level and thereby to deduce how it was intended to work.

Once the architecture of the application is understood, the construction of the system architectural model is 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 (for example, UML).

To get some understanding of the relative size of the effort involved, we note that the core of the MAF system itself is on the order of 30,000 lines of Java code (and we're not sure that we've counted everything). The system architectural model that we built to describe it is 448 lines of code. In addition, we added 369 lines of code to implement a backup dumper for the MAF system's data, 231 lines to handle adaptive method selection for image file loading, and about 500 lines of code to handle a variety of minor issues (such as bypassing the interactive "login" part of the MAF program when AWDROT restarts the program). In total the system architectural model and all support code amounted to 2,199 lines of code or about 8 percent of the target system.

Finally, both the coding effort in building

the system architectural model and the run-time overhead incurred by architectural differencing are directly proportional to the level of detail in the system architectural model. There is a trade-off to be made between this level of detail in the system architectural model and the degree of coverage that AWD RAT provides. A relatively coarse model takes proportionally less effort to build and imposes less run-time overhead but provides fewer guarantees of coverage. But perhaps this lower degree of coverage is all that is required. At the moment, we have no way to assess a priori how much coverage is enough; in practice, one designs the system architectural model in an incremental and evolutionary process, adding detail until the coverage seems adequate while the overhead remains acceptable.

What General Solutions Are Still Lacking?

A basic assumption underlying the AWD RAT methodology is that the modularity of the application code follows that of its architecture. This means that we'd like it to be the case that any significant event in the system architecture corresponds to an observable event in the code; for Java (and most other languages) this means that any significant architectural event should occur at a method boundary so that a wrapper can be used to observe the entry (or exit) from that method. In the MAF system, this was almost always the case; but there were cases in which an architecturally significant event was buried deep within a block of inline code that was not observable through the use of wrappers. In these cases, the AWD RAT tool set provides no general-purpose solution; in practice, we either rewrote the code (since we had access to the source code) or simply abandoned the attempt to observe such events.

A closely related issue is that sometimes it is necessary to distinguish the context of an event; a call to a particular method from one place in the code may be interesting, while a similar call from another location might be irrelevant or at least interpreted differently. In practice, we dealt with these issues in an ad hoc manner, capitalizing on the fact that we could instrument the "plumbing" code that connects the Java wrappers to the system architectural model. A more general solution would involve a system-modeling language that includes a richer notion of system state and state transition and a generator that produces an execution monitor that tracks the system state and that interprets the raw events delivered by the wrappers in the context of the system state.

A final issue that we encountered is that, al-

though AWD RAT can capitalize on the availability of multiple methods for accomplishing a task, it isn't often easy to find such alternative methods. Some application domains that have been systematically explored have many alternative libraries for common tasks; but in the domain explored, we found few such examples of the diversity we'd expect to see.

Future Self-Protective and Self-Regenerative Systems

AWD RAT represents one way to provide for the protection and self-healing of a critical application. AWD RAT gains its power by representing what the application should be doing and comparing this to what it actually does. However, there are other complementary techniques, including strong barriers against intrusion, intrusion detection, and variability.

The first of these involves the construction of new, inherently safer operating systems and language environments and new hardware. The combination of these will close off many of the current routes to penetrating a system (for example, buffer overflows and similar attacks). However, there will always be routes for attackers to gain and escalate their privileges; insider threats, password guessing, "social engineering attacks" (that is, fooling authorized users into extending privileges to unauthorized users), and the like will be available as means of penetration for a long time. Therefore, it will be necessary to also provide techniques for tracking possible intrusions and for making it harder for an attacker who has gained some privileges to escalate those privileges to a level where they can do serious damage.

In the long term, we imagine that these techniques will be merged in a common framework that, like AWD RAT, is driven by models that guide the synthesis of the actual protective and healing machinery; however, in this more extensive environment there will be models and synthesis tools that deal with these other aspects of the problem. For example, vulnerability analysis (Shrobe 2002) can be used to develop models of how attackers might attempt to penetrate a system and escalate their privileges. In effect, these are models of abstract plans that can be coupled with plan recognition technology to maintain a constantly evolving estimate of likely it is that the system has been penetrated (Doyle et al. 2001b, 2001a). These estimates could enable proactive deployment of protections that are geared to the observed attack vector.

Part of the AWD RAT framework involves the idea that a system should have more than one method for accomplishing each of its major

tasks. As we mentioned above, it isn't always easy to find natural diversity in software. However, there have been a number of examples of systems that induce artificial diversity (Just and Cornwell 2004) (for example, by adding random padding to the stack or by shuffling the entry vector to a library). By using models of the places in a system that are subject to "artificial diversification" together with models of the threat level the system is facing, a future AWD RAT-like system could adaptively vary the amount and kind of diversity that is introduced in order to thwart an attacker.

Finally, we imagine a software design environment in which the system architectural model is captured early (and modified continuously) in the process of designing the software. The system architectural model that we use in AWD RAT is actually "executable specifications" that can help designers explore possible different structures for their systems and that can also help detect design errors long before actual coding begins; in some cases, code can be generated automatically from the models. But in any event, the codesign of system models and the actual code is a valuable new methodology that is gaining significant attention in the software engineering community. If this trend continues, it means that in the future system models that are useful for our approach of "architectural differencing" will be available to us for free.

Conclusions

AWD RAT is an infrastructure to which an application system may be tethered in order to provide survivability properties such as error detection, fault diagnosis, backup, and recovery. It removes the concern for these properties from the domain of the application design team, instead providing these properties as infrastructure services. It uses cognitive techniques to provide the system with the self-awareness necessary to monitor and diagnose its own behavior. This frees application designers to concentrate on functionality instead of exception handling (which is usually ignored in any case).

AWD RAT's approach is cognitive in that it provides for self-awareness through a system architectural model that is a nonlinear plan and through the use of wrappers and plan-monitoring technology. Discrepancies between intended and actual behavior are diagnosed using model-based diagnosis techniques, while recovery is guided by decision-theoretic methods.

We have demonstrated the effectiveness of

AWD RAT in detecting, containing, and recovering from compromises that might arise from a broad variety of attack types. AWD RAT is not particularly concerned with the attack vector, since its major source of power is that it has a model of what the application should be doing rather than a library of specific attack types.

Acknowledgments

This article describes research conducted at the 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 Number FA8750-04-20240. The views presented are those of the author alone and do not represent the view of DARPA or AFRL.

Notes

1. In principle, AWD RAT could be applied to operating system code, but we have never experimented with doing so.
2. In this case, the second library is a widely available, open source library that we included as part of the AWD RAT instrumentation of the target system.
3. There are other possible notions that one might want to optimize. In particular, some people are more risk averse than others. To accommodate this, it's possible to use a different objective function, for example, one that uses a weighted sum of expected benefit and expected cost where the weights represent an assessment of the relative importance of avoiding negative outcomes versus achieving the highest possible benefits.

References

- Allen, J.; Christie, A.; Fithen, W.; McHugh, J.; Pickel, J.; and Stoner, E. 2000. State of the Practice of Intrusion Detection Technologies. Technical Report, Networked Systems Survivability Program, CMU/SEI-99-TR-028, CMU Software Engineering Institute, Pittsburgh, PA.
- Axelsson, S. 1998. Research in Intrusion-Detection Systems: A Survey. Technical Report 98-17, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden.
- Balzer, R., and Goldman, N. 2000. Mediating Connectors: A NonBypassable Process Wrapping Technology. In *Proceedings of the First Darpa Information Security Conference and Exhibition (DISCEX-II)*, Volume II, 361-368. Los Alamitos, CA: IEEE Computer Society.
- Bobrow, B.; DeMichiel, D.; Gabriel, R.; Keene, S.; Kiczales, G.; and Moon, D. 1988. Common Lisp Object System Specification. *SIGPLAN Notices* 23 (September).
- Debar, H.; Dacier, M.; and Wespi, A. 1999. Towards a Taxonomy of Intrusion-Detection Systems. *Computer Networks* 31(8): 805-822.
- deKleer, J., and Williams, B. 1989. Diagnosis with Behavior Modes. In *Proceedings of the Eleventh Interna-*

tional Joint Conference on Artificial Intelligence. San Francisco: Morgan Kaufmann.

Doyle, J.; Kohone, I.; Long, W.; Shrobe, H.; and Szolovits, P. 2001a. Agile Monitoring for Cyber Defense. In *Proceedings of the Second DARPA Information Security Conference and Exhibition (DISCEX-II)*. Los Alamitos, CA: IEEE Computer Society.

Doyle, J.; Kohone, I.; Long, W.; Shrobe, H.; and Szolovits, P. 2001b. Event Recognition Beyond Signature and Anomaly. In *Proceedings of the Second IEEE Information Assurance Workshop*. Los Alamitos, CA: IEEE Computer Society.

Hollebeek, T., and Waltzman, R. 2004. The Role of Suspicion in Model-Based Intrusion Detection. Paper presented at the 2004 Workshop on New Security Paradigms, White Point Beach Resort, Nova Scotia, Canada, 20–23 September.

Just, J. E., and Cornwell, M. 2004. Review and Analysis of Synthetic Diversity for Breaking Monocultures. In *WORM '04: Proceedings of the 2004 ACM Workshop on Rapid Malcode*, 23–32. New York: ACM Press.

Keene, S. 1989. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Reading, MA: Addison-Wesley.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; and Griswold, W. G. 2001. An Overview of AspectJ. In *Proceedings of the Fifteenth European Conference on Object-Oriented Programming*, Lecture Notes in Artificial Intelligence, Volume 707, 327–353. Berlin: Springer.

Laddaga, R.; Robertson, P.; and Shrobe, H. E. 2001. Probabilistic Dispatch, Dynamic Domain Architecture, and Self-Adaptive Software. In *Self-Adaptive Software*, ed. R. Laddaga, P. Robertson, and H. Shrobe, 227–237. Berlin: Springer-Verlag.

Lunt, T. F. 1993. A Survey of Intrusion Detection Techniques. *Computer Security* 12(4): 405–418.

Rich, C. 1981. Inspection Methods in Programming. MIT AI Lab Technical Report 604, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Rich, C., and Shrobe, H. E. 1976. Initial Report on a Lisp Programmer's Apprentice. MIT AI Lab Technical Report 354, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Shrobe, H. 1979. Dependency-Directed Reasoning for Complex Program Understanding. MIT AI Lab Technical Report 503, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Shrobe, H. 2001. Model-Based Diagnosis for Information Survivability. In *Self-Adaptive Software*, ed. R. Laddaga, P. Robertson, and H. Shrobe. Berlin: Springer-Verlag.

Shrobe, H. 2002. Computational Vulnerability Analysis for Information Survivability. In *Proceedings of the Fourteenth Innovative Applications of Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Howard Shrobe is a principal research scientist at the Massachusetts Institute of Technology (MIT) Artificial Intelligence Laboratory. He received his M.S. and



Ph.D. from MIT in 1975 and 1978 and his B.S. from Yale College in 1968. He has been at the MIT AI Lab since arriving in 1973 as a graduate student. He also worked at Symbolics Inc., serving as a technical director and vice president of technology. He served as chief scientist of the Defense Advanced Research Project's Information Technology Office for three years. His e-mail address is hes@ai.mit.edu.

Robert Laddaga is a research scientist at the Massachusetts Institute of Technology (MIT) Artificial Intelligence Laboratory. He has also served as a program manager at DARPA in the then Information Technology Office and as project director of the MIT AI Lab Intelligent Information Infrastructure Project, which developed software and systems for intelligent access to e-mail and web-based publications. He also managed the transition of publication server technology to the executive office of the president of the United States. Since 1999 he has worked on projects on self-adaptive software, smart spaces, information survivability, and middleware and applications for networked sensor ensembles.

Robert Balzer received his B.S., M.S., and Ph.D. degrees in electrical engineering from the Carnegie Institute of Technology, Pittsburgh, Pennsylvania, in 1964, 1965, and 1966, respectively. After several years at the Rand Corporation, he left to help form the University of Southern California's Information Sciences Institute (USC-ISI), where he served as director of ISI's Software Sciences Division and professor of computer science at USC. In 2000 he joined Teknowledge Corporation as its chief technical officer and director of its distributed systems unit. The distributed systems unit combines artificial intelligence, database, and software engineering techniques to automate the software development process. His current research includes wrapping COTS products to provide safe and secure execution environments, extend their functionality, and integrate them together; instrumenting software architectures; and generating systems from domain-specific specifications.

Neil Goldman is affiliated with Teknowledge Corporation's Distributed Systems business unit in Marina Del Rey, California.

Dave Wile is a senior research scientist with Teknowledge Corporation's Distributed Systems business unit in Marina Del Rey, California. He received his Sc.B degree from Brown University in applied mathematics in 1967, and a Ph.D. degree in computer science from Carnegie Mellon University in 1974.

Marcelo Tallis is affiliated with Teknowledge Corporation's Distributed Systems business unit in Marina Del Rey, California.

Tim Hollebeek is a research scientist with Teknowledge Corporation.

Alexander Egyed is affiliated with Teknowledge Corporation's Distributed Systems business unit in Marina Del Rey, California.