AWDRAT: A Cognitive Middleware System for Information Survivability [*]

Howard Shrobe and **Robert Laddaga**
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

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

Abstract

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 paper describes AWDRAT, a middleware system for providing survivability to both new and legacy applications. AWDRAT stands for Architectural-differencing, Wrappers, Diagnosis, Recovery, Adaptive software, and Trust-modeling. AWDRAT 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, AWDRAT conducts a diagnosis that figures out 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.

AWDRAT has been used on an example application system, a graphical editor for constructing mission plans. We present data showing the effectiveness of AWDRAT in detecting a variety of compromises to the application system.

Overview

To the extent that traditional systems provide for immunity against attack, they rely either on detecting known patterns of attack or on detecting statistically anomalous behavior. Neither of these approaches is satisfactory: The first approach fails in the face of novel attacks, producing an unacceptably high false negative rate. The second approach confounds unusual behavior with illegal behavior; this produces unacceptably high false positive rates and lacks diagnostic resolution even when and intrusion is correctly flagged.

In this paper, we present AWDRAT, a middleware system to which an existing application software may be retrofitted and that provides immunity to compromises of the 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, Recovery, Adaptivity and Trust-modeling; it provides a variety 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 within the AWDRAT environment behave adaptively, and with the aid of AWDRAT these system regenerate themselves when attacks cause serious damage.

The AWDRAT Architecture

The AWDRAT architecture is shown in Figure 1. AWDRAT is provided with models of the intended behavior of the applications it is intended to protect. These models are based on a “plan level” decomposition that provides pre- and post- and invariant conditions for each module. AWDRAT actively enforces these declarative models of intended behavior using “wrapper” technology. Non-bypassable wrappers check the model's conditions at runtime, allowing execution to proceed only if the observed behavior is consistent with the model's constraints. We call this technique “Architectural Differencing”. In the event that unanticipated behavior is detected, AWDRAT 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 update a “trust model” indicating the likelihood and types of compromise that may have been effected to each computational resource. Finally, AWDRAT helps the application recover from failure, using this trust model to guide its selection of computational techniques (assuming that the application has more than one method for carrying out its intended tasks) and in its selection of computational resources to be used in completing the task.

AWDRAT uses its model of an application's intended behavior to recognize the critical data that must be preserved in case of failure. AWDRAT generates wrappers that dynamically provision backup copies and redundant encodings of this critical data. During recovery efforts, AWDRAT installs these backup copies in place of compromised data resources.

Using this combination of technologies, AWDRAT provides “cognitive immunity” to both intentional and accidental compromises. An application that runs within the AW-

[*]This article describe 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 FA8750-04-2-0240. The views presented are those of the author alone and do not represent the view of DARPA or AFRL.

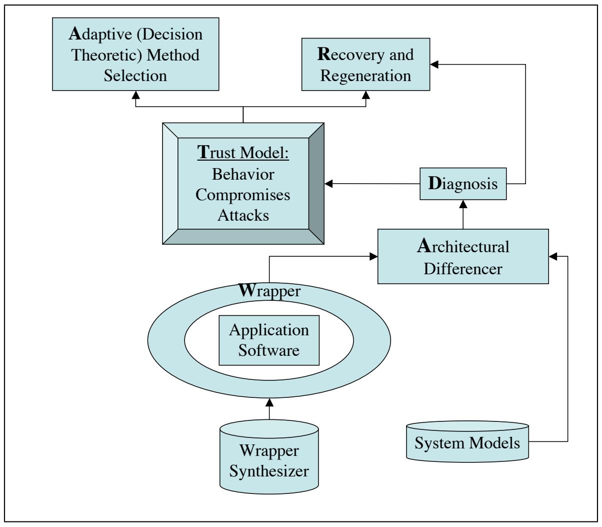
Adaptive (Decision Theoretic) Method Selection
Recovery and Regeneration
Trust Model: Behavior Compromises Attacks
Diagnosis
Architectural Differencer
Wrapper
Application Software
Wrapper Synthesizer
System Models

Figure 1: The AWDRAT Architecture

DRAT 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.

Synthesis of Wrappers and Execution Model

AWDRAT, in fact, employs two distinct wrapper technologies: SafeFamily(Balzer & Goldman 2000; Hollebeek & Waltzman 2004) and JavaWrap. The first of these encapsulates system DLL's, allowing AWDRAT 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 to invoke the original method if desired. 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. 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 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 AWDRAT from a "System Architectural Model" such as that shown in Figure

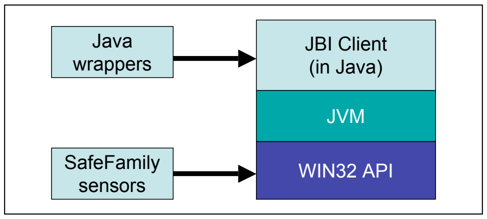

Figure 2: Two Types of Wrappers Used in AWDRAT

3. The model is written in a language similar to the "Plan Calculus" of the Programmer's Apprentice (Rich & 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 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). Each component of the 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 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 AWDRAT wrapper synthesizer collects up all event specifications used in the model and then synthesizes the wrapper method code and the two required XML specification files as is shown in Figure 4.

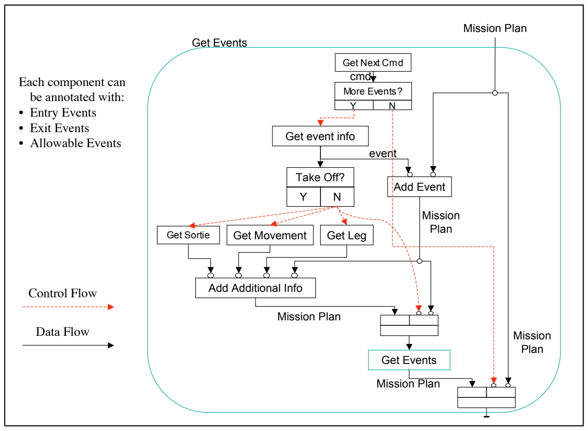

Figure 3: An Example System Model

Architectural Differencing

In addition to synthesizing wrappers, the AWDRAT generator also synthesizes an "execution monitor" corresponding to the system 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

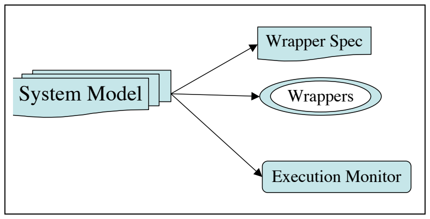

Figure 4: Generating the Wrapper Plumbing

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 AWDRAT run-time system, are 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 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 model 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 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 sub-network (if it has one) and propagate 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 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:

- 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.
- 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 post-conditions are checked and diagnosis is invoked if the check fails.
- Otherwise the event is checked to see if its 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, i.e. 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, AWDRAT detects any deviation of the application from the abstract behavior specified in its System Architectural Model and invokes its diagnostic services.

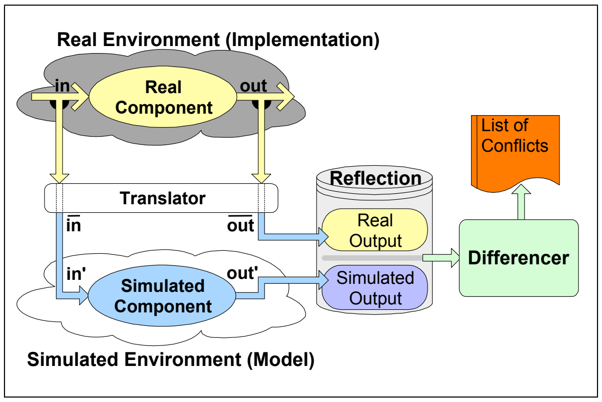

Figure 5: Architectural Differencing

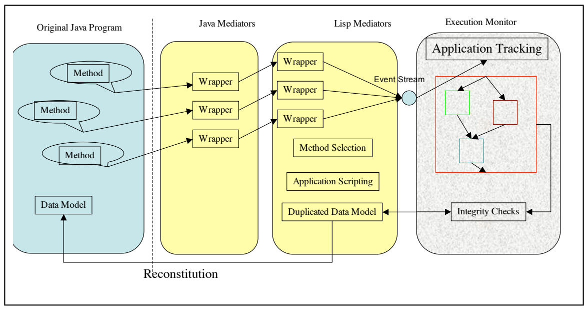

Figure 6: The Generated Plumbing

Diagnostic Reasoning

AWDRAT's diagnostic service is described in more detail in (Shrobe 2001) and draws heavily on ideas in (deKleer & Williams 1989). 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 models for faulty behavior. As just 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. As the execution monitor does this, it 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 7. This is similar to the reasoning techniques in (Shrobe 1979).

Should a discrepancy between actual and intended behavior be detected, this will show up as a contradiction in the database of assertions describing the application's execution

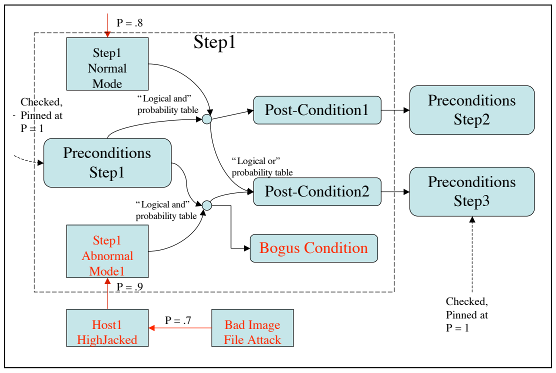

Figure 7: Dependency Graph

history. Diagnosis then consists of finding alternative behavior models for some subset of the components in the architectural model such that the contradiction disappears when these models 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 8. 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.

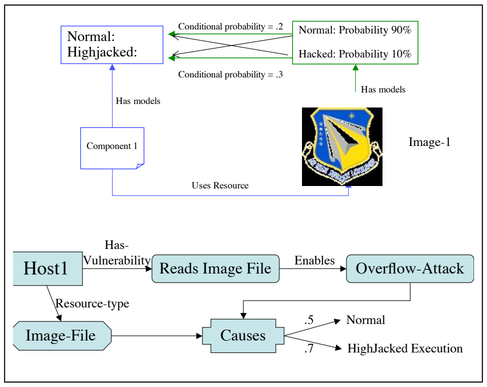

Figure 8: Diagnosis With Fault and Attack Models

Self-Adaptive Software

Recovery in AWDRAT depends critically on self-adaptive techniques such as those described in (Laddaga, Robertson, & 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 AWDRAT. This application loads image files (e.g. GIF, JPEG) and, as it happens, there is a vulnerability (since fixed) related to loading malformed image files. This is enabled by the use of a "native library" (i.e. 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 "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 9, 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.

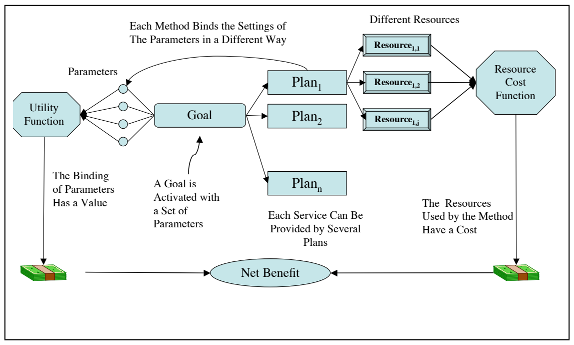
Each Method Binds the Settings of The Parameters in a Different Way
Different Resources
Parameters
Utility Function
Goal
Plan1
Plan2
Plann
Resource1,1
Resource1,2
Resource1,j
Resource Cost Function
The Binding of Parameters Has a Value
A Goal is Activated with a Set of Parameters
Each Service Can Be Provided by Several Plans
The Resources Used by the Method Have a Cost
Net Benefit

Figure 9: Adaptive Software Picks the Best Method

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 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. 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). 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.

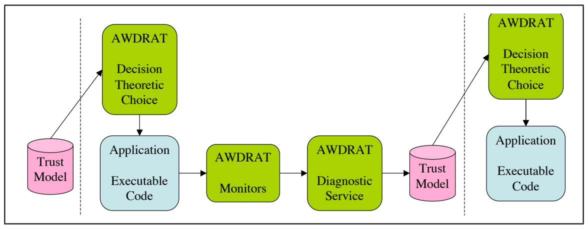

Figure 10: The Trust Model Guides Future Recovery

Experimentation and Results

AWDRAT's goal is to guarantee that the application tethered to it faithfully executes the intent of the software designer; for example, for an interactive system this means that the system should faithfully execute the commands specified by its user (e.g 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 AWDRAT we applied it to the defense of a particular application system, the MAF interactive mission planner - a component of of the DARPA Demval demonstration system which is in turn based on Rome Labs Joint Battlesphere Infrastructure (JBI). MAF is a Java-based program whose primary vulnerabilities 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. AWDRAT'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, AWDRAT focuses on the maintenance of data integrity and data privacy.

We identified the following properties that AWDRAT should guarantee:

- 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
  - All the fields requested by the GUI should be present
- The application should not be able to open a port to any application other than to the JBI servers with which it must communicate (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.

Experimental Methodology

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).

The core AWDRAT 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 Hashtable for the overall mission data and four additional Hashtables 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 hashtable 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 (e.g. "setInformation", "getInformation"). In addition, Java defines methods on Hashtables (e,g, "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 (e.g. using the Hashtable 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 (AWDRAT 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 attacks 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.

In more detail, here are the tests conducted by category:

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
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
3. Wrapped Methods (used only for "loadImage")
   - Make call to write files
   - Make call to launch external application
   - Make call to open port

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 only 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 an 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, Awdrat 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-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 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 (e.g. 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

All attempts to launch an application, write a file other than those sanctioned or to open a 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 AWDRAT 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 exact same API call. This only occurs 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 request 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, AWDRAT's machinery 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 AWDRAT had

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%

Figure 11: Experimental Results

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 AWDRAT execution monitor isn't aware of thread structure.

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, AWDRAT does restart the application and rebuild its data structures in these cases as well. For the Wrapped cases (i.e. those involving simulated corrupt image files) the last column is listed because the dominant diagnostic hypothesis in those cases is that a attack was launched from payload code embedded in the image being loaded. In these cases, switching to the Pure Java method and/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 (and/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 model is a reasonable abstraction of the program.

In addition to these extensive internal tests, we also subjected AWDRAT 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 AWDRAT was not expected to deal with. Nevertheless AWDRAT performed at a level above the programmatic goals of the DARPA SRS program that sponsored this effort.

Moving Towards Practicality

AWDRAT is not yet ready for full deployment, we are forming plans to apply it to another set of target application and server systems. There are two major costs involved in the use of AWDRAT: The development cost of building a system model and the runtime overhead that AWDRAT imposes on the hosted application system. In our experience so far, the second of these costs is negligible. Since we used AWDRAT to defend an interactive application, the key question is whether the overhead slows down the user interface and we experienced no observable degradation of performance. However, for real-time embedded applications we will need to be much more careful about containing the cost of execution monitoring.

The development cost of building the system model is also manageable in our experience if the application's architecture is well understood. This wasn't true for the MAF system, being demonstration prototype code its architecture was not documented and we had to engage in "software archeology" to gain an understanding. One of the key tools we used for this effort was AWDRAT's wrappers, which allowed us to trace the execution of the application system at the method call level and thereby deduce how it was intended to work.

We have also learned that there is a tradeoff to be made between the level of detail in the system and the degree of coverage that AWDRAT provides. A relatively coarse model, takes proportionally less effort to build and imposes less runtime overhead but provides fewer guarantees of coverage. In the best of cases, the application to be protected is well understood and reasonably well modularized. In these cases, the construction of the system model is straightforward (and might be generated from existing architectural models such as UML diagrams). We imagine that in practice the system model will be constructed by software engineers familiar with the application system with modest help from the designers of AWDRAT.

Conclusions and Future Work

AWDRAT 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).

AWDRAT's approach is cognitive in that it provides for self-awareness through a system model that is a non-linear 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 AWDRAT in detecting, containing and recovering from compromises that might arise from a broad variety of attack types. AWDRAT 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.

The current demonstration has focussed on a client application, but we are currently building a system model for a open-source, Java based version of a Domain Name Server and we will soon test whether AWDRAT will provide survivability and resilience to attacks for such a server application.

References

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

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

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

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

Keene, S. 1989. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Number ISBN 0-201-17589-4. 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 15th European Conference on Object-Oriented Programming*, 327–353.

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

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

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

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

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