

# Computational Vulnerability Analysis for Information Survivability

*Howard Shrobe*

■ The infrastructure of modern society is controlled by software systems. These systems are vulnerable to attacks; several such attacks, launched by “recreation hackers,” have already led to severe disruption. However, a concerted and planned attack whose goal is to reap harm could lead to catastrophic results (for example, by disabling the computers that control the electrical power grid for a sustained period of time). The survivability of such information systems in the face of attacks is therefore an area of extreme importance to society. This article is set in the context of self-adaptive survivable systems: software that judges the trustworthiness of the computational resources in its environment and that chooses how to achieve its goals in light of this trust model. Each self-adaptive survivable system detects and diagnoses compromises of its resources, taking whatever actions are necessary to recover from attack. In addition, a long-term monitoring system collects evidence from intrusion detectors, firewalls, and all the self-adaptive components, building a composite trust model used by each component. Self-adaptive survivable systems contain models of their intended behavior; models of the required computational resources; models of the ways in which these resources can be compromised; and finally, models of the ways in which a system can be attacked and how such attacks can lead to compromises of the computational resources.

In this article, I focus on computational vulnerability analysis: a system that, given a description of a computational environment, deduces all the attacks that are possible. In particular, its goal is to develop multistage attack models in which the compromise of one resource is used to facilitate the compromise of other, more valuable resources. Although the ultimate aim is to use these models online as part of a self-adaptive system, there are other offline uses as well that we are deploying first to help system administrators assess the vulnerabilities of their computing environment.

The infrastructure of modern society is controlled by computational systems that are vulnerable to information attacks. A skillful attack could lead to consequences as dire as those of modern warfare. There is a pressing need for new approaches to protect our computational infrastructure from such attacks and enable it to continue functioning even when attacks have successfully been launched.

The premise of my group’s research is that to protect the infrastructure we need to restructure these software systems as *self-adaptive survivable systems*. Such software systems must be informed by a trust model that indicates which resources are to be trusted. When such a system starts a task, it chooses the method that the trust model indicates is most likely to avoid compromised resources. In addition, such a system must be capable of detecting its own malfunction, it must be able to diagnose the failure, and it must be capable of repairing itself after the failure. For example, a system might notice through self-monitoring that it is running much slower than expected. It might, therefore, deduce that the scheduler of the computer it is running on has been compromised and that the compromise resulted from the use of a buffer-overflow attack that gained root access to the system and used this privilege to change the scheduler policy. The buffer-overflow attack in turn might have exploited a vulnerability of a web server (as, for example, happened in the “code-red” worm attack). Given this diagnosis, the trust model should be updated to indicate that the computer’s operating system was compromised and should be avoided in the future if possible. Techniques for this type of diagnosis are described in Shrobe (2001).

The trust model is also influenced by collating evidence from many available sources over a long period of time. In our lab, for example, we notice several alerts from our intrusion-detection system over a couple a days, which was followed by a period in which nothing anomalous happened. However, then we began to notice that the consumption of disk space and the amount of network traffic from outside the lab were increasing, which continued for some time. The load then leveled off. What had happened is that a user password had been stolen and that a public ftp site had been set up for the use of the friends of the password thief. This incident is an instance of a common attack plan. Such attack plans have multiple stages, temporal constraints between the stages, and constraints within each stage on values and their derivatives (for example, the rate of growth of disk space consumption). These can, therefore, be used as *trend templates* for collating and analyzing the alerts from intrusion-detection systems and the data in common system logs. Trend template analysis provides perspective over a longer period of time than the intrusion-detection systems themselves possess, allowing detection of attacks that are intentionally subtle. Long-term monitoring systems capable of conducting trend template-driven attack plan recognition are described in Doyle et al. (2001a, 2001b).

Trust modeling thus depends both on attack plan recognition as well as on the self-diagnosis of self-adaptive software systems. The resulting trust model includes models of what computational resources have been compromised, what attacks were used to effect this attack, and what vulnerability was exploited by the attack. Key to all these tasks is having a comprehensive set of attack models.

This article focuses on *computational vulnerability analysis*, a systematic method for developing attack models used both in attack plan recognition and self-diagnosis of adaptive systems. All current systems are driven either by signatures of specific exploits (for example, the telltales of a password scan) or anomaly profiling (for example, detecting a difference in behavior between the current process and a statistical norm). Neither of these methods alone is capable of dealing with a skillful attacker who would stage his/her attack slowly to avoid detection, would move in stages, and would use a compromise at one stage to gain access to more valuable resources later on. The systematic nature of computation vulnerability analysis and the use of its attack plans in both long-term monitoring and self-diagnos-

ing adaptive systems leads to increased precision in trust modeling and greater survivability of critical systems.

## Contributions of This Work

My group has developed a model-based technique, which we call *computational vulnerability analysis*, for analyzing vulnerabilities and attacks. Rather than relying on a catalog of known specific attacks, we instead reason from first principles to develop a much more comprehensive analysis of the vulnerabilities. Furthermore, the attacks developed in this analysis include both single-stage attacks as well as multistage attacks. These issues are crucial when failure is caused by a concerted attack by a malicious opponent who is attempting to avoid detection.

We developed a unified framework for reasoning about the failures of computations and about these failures and how they are related to compromises of the underlying resources, the vulnerabilities of these resources, and the method by which these vulnerabilities enable attacks. We then extended previous work in model-based diagnosis (Davis and Shrobe 1982; deKleer and Williams 1989, 1987; Hamscher and Davis 1988; Srinivas 1995) to enable systems capable of self-diagnosis, recovery, and adaptation. We used this framework to build long-term monitoring systems (Doyle et al. 2001a, 2001b) capable of attack plan recognition. Both attack plan recognition and self-diagnosis lead to updated estimates of the trustability of the computational resources. These estimates, which form the trust model, inform all future decision making about how to achieve goals.

In addition to its role in survivable systems, computational vulnerability analysis can also be used offline to assess the vulnerability of and identify weak links in a computational environment. This use can help system administrators improve the security and robustness of their network, often by instituting simple changes. My group is currently using the system, in a limited way, to assess the vulnerabilities of our lab's computing environment; as the system matures, we plan to apply it more systematically to the entire lab. We are also in the process of connecting the computation vulnerability analysis system to our long-term monitoring system and connecting the monitoring system to a commercial intrusion detector. We plan to begin deploying this monitoring system within the next six months.

This article first describes the modeling framework and reasoning processes used in computational vulnerability analysis and shows its application to a small section of our

lab's computing environment. I conclude by explaining how attack plans fit within the self-diagnostic and the long-term monitoring frameworks.

## Computational Vulnerability Analysis

In this section, I examine the core issue of this article, which is how to make the modeling of attacks and vulnerabilities systematic.

I do this examination by grounding the analysis in a comprehensive ontology that covers system properties, system types, system structure, and the control and dependency relationships between system components.

This ontology covers what types of computing resources are present in the environment, how the resources are composed from components (for example, an operating system has a scheduler, a file system, and so on), how the components control one another's behavior, and what vulnerabilities are known to be present in different classes of these components. Finally, the models indicate how desirable properties of such systems depend on the correct functioning of certain components of the system (for example, predictable performance of a computer system depends on the correct functioning of its scheduler).

A relatively simple reasoning process (encoded in a rule-based system) then explores how a desirable property of a system can be impacted (for example, you can impact the predictability of performance by affecting the scheduler, which in turn can be done by changing its input parameters, which in turn can be done by gaining root access, which finally is enabled by a buffer-overflow attack on a process running with root privileges). The output of this reasoning is a set of multistage attacks, each of which is capable of affecting the property of interest.

I also provide a structural model of the entire computing environment under consideration, including the following:

**Network structure and topology:** How is the network decomposed into subnets? Which nodes are on which subnets? Which routers and switches connect the subnets? What types of filters and firewalls provide control of the information flow between subnets?

**System types:** What type of hardware is in each node? How is the hardware decomposed into subsystems? What type of operating system is in each node? How is the operating system decomposed into subsystems?

**Server and user software suites:** What software function is deployed on each node?

If the goal is to affect the reliable-performance property of some component? $x$   
 Then find a component  $?y$  of  $?x$  that contributes to the delivery of that property and find a way to control  $?y$

Figure 1. A Vulnerability Analysis Rule Dealing with Control.

**Access rights:** What are the access rights to data and how are they controlled?

**Data storage:** What are the places in which data are stored or transmitted?

The next step is to model dependencies. I begin with a list of desirable properties that the computational resources are supposed to deliver. Typical properties include reliable performance, privacy of communications, Integrity of communications, integrity of stored data, and privacy of stored data.

Within the diagnostic framework, each such property corresponds to a normal behavioral mode of some (or several) computational resource(s). For example, reliable computational performance is a property to which the scheduler contributes, but data privacy is a property contributed by the access-control mechanisms.

### Control Relationships

I now turn attention to a rule base that utilizes this ontology to reason about how one might affect a desirable property. The goal is to make this rule base as abstract and general as possible. For example, see figure 1. This abstract rule is a paraphrase of the actual rule, which is coded in a Lisp-based rule system).

This rule (figure 1) puts the notion of control and dependency at the center of the reasoning process. There are several rules about how to gain control of components, which are quite general. Figure 2 contains examples of such general and abstract rules.

At the leaves of this reasoning chain is specific information about vulnerabilities and how to exploit them. For example, Microsoft IIS web servers below a certain patch level are vulnerable to buffer-overflow attacks, and buffer-overflow attacks are capable of taking control of the components that are vulnerable.

One of the rules shown previously indicates that one can control a component by modifying its input (figure 2). Figure 3 describes how an input can be controlled.

If the goal is to control a component ?x  
Then find an input ?y to ?x  
and find a way to modify ?y

If the goal is to control a component ?x  
Then find a component ?y of ?x  
and find a way to control ?y

If the goal is to control a component ?x  
Then find a vulnerability ?y  
of the component ?x  
and find a way to exploit ?y  
to take control of ?x.

Figure 2. Vulnerability Rules Dealing with Controlling Input and Components.

If the goal is to modify an input ?x of component ?y  
then find a component ?z which controls the input ?x  
and find a way to gain control of ?z

If the goal is to modify an input ?x of component ?y  
then find a component ?z of the input ?x  
and find a way to modify ?z

Figure 3. Rules about Controlling Input.

If the goal is to gain access to operation ?x on object ?y  
and operation ?x on ?y requires capability ?z subsume ?z  
and find a way to take control of ?p.

If the goal is to gain access to operation ?x on object ?y  
and operation ?x on ?y requires capability ?z  
then find a user ?u whose capability ?w subsumes ?z  
and find a way to log in as ?u  
and launch a process ?p with capability ?w

Figure 4. Rules about Access Control.

## Access Rights

Within most computer systems, the ability to read or modify data depends on obtaining access rights to the data. In my group, we model access rights in a more general way than is used in many actual systems:

First, for each type of object, we enumerate the operations that can be performed on objects of that type.

Second, for each operation, we specify the capabilities that are required to perform the operation.

Third, the capabilities are related by a subsumption relationship that forms a directed acyclic graph.

Fourth, for each agent (that is, a user or a process), we enumerate the capabilities that the agent possesses at any time.

Fifth, an agent is assumed to be able to perform an operation on an object only if it possesses a capability at least as strong as that required for the operation.

Sixth, typically, groups of machines manage access rights collectively (for example, work groups in Microsoft WINDOWS, NIS in UNIX environments). We refer to such a collection of machines as an *access pool*.

Seventh, the structure of access pools can be orthogonal to the network topology. Machines in different subnets can be parts of the same access pool, and machines on a common subnet can be members of different access pools.

Given this framework, we provide rules that describe how to gain access to objects in figure 4.

## Knowledge of Secrets

Logging on to a system typically requires knowledge of a secret (for example, a password). A set of rules describes how to obtain knowledge of a password:

First, to obtain knowledge of a password, find it by guessing, using a guessing attack.

Second, to obtain knowledge of a password, sniff it. To sniff a piece of data, place a parasitic virus on the user's machine. To sniff a piece of data, monitor network traffic that might contain the datum. To sniff a piece of data, find a file containing the data and gain access to it.

Third, to obtain knowledge of a password, gain write access to the password file and change it.

## Network Structure

The next section of rules deals with networks. As mentioned previously, networks are described in terms of the decomposition into subnets and the connections of subnets by routers and switches. In addition, for each subnet, I provide a description of the media type; some

subnets are shared media, for example, coaxial-cable-based ethernet and wireless ethernet. In such subnets, any connected computer can monitor any of the traffic. Other subnets are switched media (for example, 10, 100, and 1000 base-T-type ethernet); in these networks, only the switch sees all the traffic (although it is possible to direct the switch to reflect all traffic to a specific port). Switches and routers are themselves computers that have presence on the network, which means that like any other computer, there are exploits that will gain control of them. However, it is typical that the switches and routers are members of a special access pool, using separate capabilities and passwords.

Given this descriptive machinery it now becomes possible to provide another rule:

To gain knowledge of some information gain the ability to monitor network traffic.

### Residences and Format Transformations

The last set of modeling issues have to do with the various places in which data live and how data are transformed between various representations. The following issues are modeled:

First, data elements reside in many places.

Second, executable code resides in many places: main memory, boot files, and paging files.

Third, data elements and code move between their various residences. Data migrations go through peripheral controllers. Data migrations go through networks.

Given these representations, we then provide the following rules:

First, to modify or observe a data element, find a residence of the element and find a way to modify or observe it in the residence.

Second, to modify or observe a data element, find a migration path, and find a way to modify or observe it during the transmission.

Further rules provide details of how one might gain control of a peripheral controller or a network segment to modify data during transmission.

For example, to control traffic on a network segment launch, use a *man in the middle* attack by gaining control of a machine on the network and then finding a way to redirect traffic to the machine rather than to the router or switch.

To observe network traffic, get control of a switch or router and a user machine and reflect the traffic to the user machine.

To modify network traffic, launch an insert-ed packet attack. Thus, get control of a machine on the network and then send a packet from the machine with the correct serial num-

ber but wrong data before the real sender sends the correct data.

A somewhat analogous issue has to do with the various formats that data and code take on and the processes that transform data and code between these formats. In particular, code can exist in at least the following formats: source, compiled, and linked executable images. In many systems, there are other representations as well (for example, JAR [JAVA archive] files for JAVA code). In addition, processes such as compilation and linking transform code between these formats, leading to the following rules:

To modify a software component, find an upstream representation of the component and then find a way to modify that representation and cause the transformation between representations to happen.

To modify a software component, gain control of the processes that perform the transformation from upstream to downstream representation.

### An Example

The following example illustrates how these representations and rules interact to analyze the vulnerabilities of a computer. Suppose we are interested in affecting the performance of a specific computer. The rule base would then generate the following plan:

First, one goal is to control the scheduler of the computer because the scheduler is a component that impacts performance.

Second, one way to do that is to modify the scheduler's policy parameters because the policy parameters are input to the scheduler.

Third, one way to modify the policy parameters is by gaining root access to the computer because root access is required to modify these parameters.

One way to gain root access is to use a buffer-overflow attack on a web server because the web server possesses root capabilities, and the web server is vulnerable to buffer-overflow attacks.

For this attack to succeed in impacting performance, every step of the plan must succeed. Each of these steps has an a priori probability based on its inherent difficulty. The analysis process must take into account not just the general strategies but also the specific features of individual machines, network segments, routers, fire walls, packet filters, and so on. The attack plans include only those that satisfy all these constraints. A computer can be vulnerable to an exploit, but if there is a firewall isolating it from the attacker, the analysis will not develop an attack plan exploiting this vulnerability.

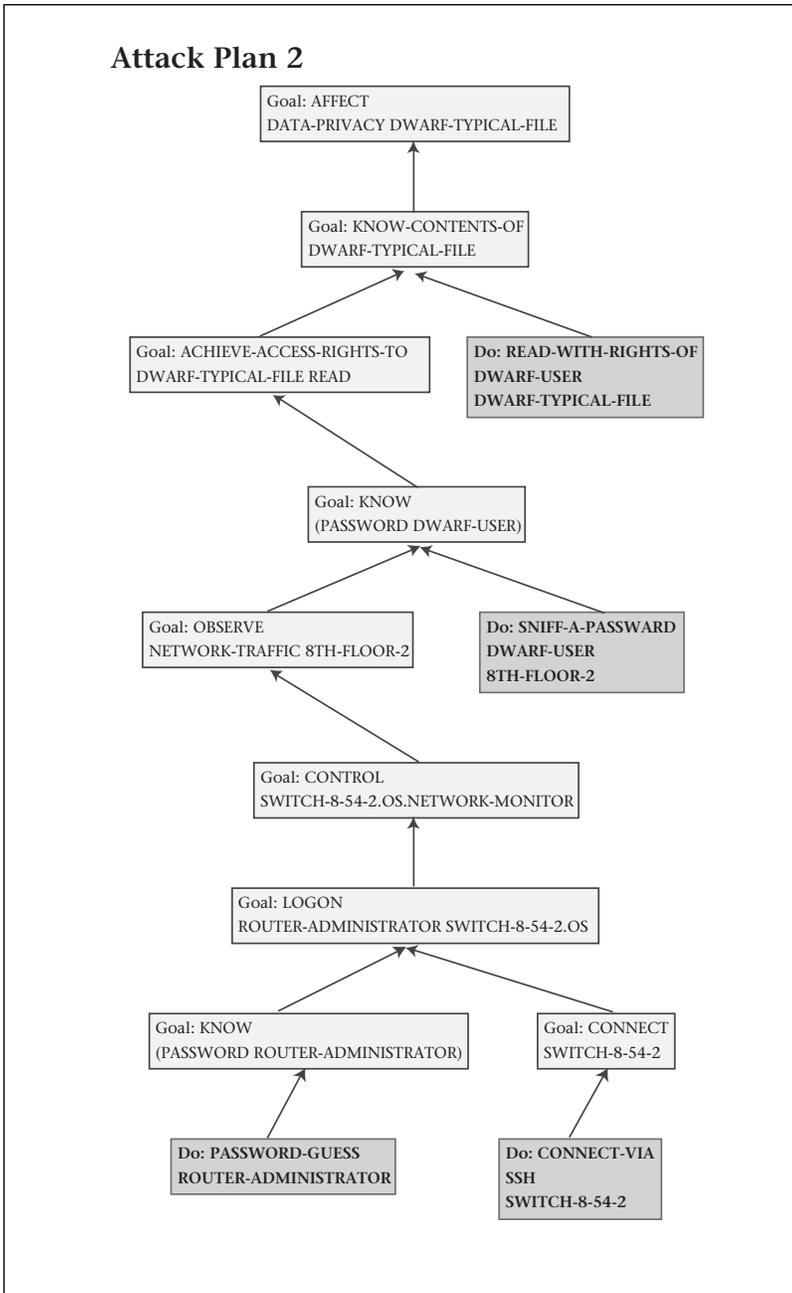


Figure 5. A Plan for Affecting Privacy.

Figures 5 and 6 show two attack plans that our system developed to attack privacy. Other plans developed are more complex. Each plan is an and-or tree (goal nodes are or nodes); they can have several incoming links from plan nodes; and all that is required is that one of the plans work. Plan nodes are and nodes; each subgoal must be fulfilled for a plan to be valid). The leaves of the tree are primitive actions, that is, actual attack steps. The figures show one slice through the and-or tree for simplicity.

For the given system description, our vulnerability analyzer generated seven attack plans for the privacy property and nine plans for attacking performance.

We now turn briefly to the question of how the attack plans are used in diagnostic reasoning and long-term monitoring. In both cases, the attack plans are transformed: For diagnostic reasoning, they are converted into components of a Bayesian network. In this form, they help explain why a computation has failed, and they also deduce what resources are therefore likely to have been compromised. For long-term monitoring, they are transformed into trend templates. In this format, they function as a timeline for how a skillful attacker would stage an assault on the analyzed network. The monitoring system accepts and collates input from intrusion detectors, firewalls, and self-monitoring applications in an attempt to detect more pernicious, multistage attacks.

We briefly describe each use in the next two sections.

### Application to Diagnosis

Figure 7 shows a model of a fictitious distributed financial system that we use to illustrate the reasoning process. The system consists of five interconnected software modules—(1) WEB-SERVER, (2) DOLLAR-MONITOR, (3) BOND-TRADER, (4) YEN-MONITOR, and (5) CURRENCY-TRADER—using four underlying computational resources—that is, the computers (1) WALLST-SERVER, (2) JPMORGAN, (3) BONDRUS, and (4) TRADER-JOE. We use computational vulnerability analysis to deduce that one or more attack types are present in the environment, leading to a three-tiered model as shown in figure 8. The first tier is the *computational level*, which models the behavior of the computation being diagnosed; the second tier is the *resource level*, which monitors the degree of compromise in the resources used in the computation; the third tier is the *attack layer*, which models attacks and vulnerabilities. In this example, we show two attack types: (1) buffer overflow and (2) packet flood. *Packet floods* can affect each of the resources because they are all networked systems; *buffer overflows* affect only the two resources that are instances of a system type that is vulnerable to such attacks.

A single compromise of an operating system component, such as the scheduler, can lead to anomalous behavior in several application components. This example illustrates a *common mode failure*; intuitively, a common mode failure occurs when a single fault (for example, an inaccurate power supply) leads to faults at

several observable points in the systems (for example, several transistors misbehave because their biasing power is incorrect). Formally, there is a common mode failure whenever the probabilities of the failure modes of two (or more) components are dependent.

We deal with common mode failures as follows: Our modeling framework includes three kinds of objects: (1) computational components (represented by a set of input-output relationships and delay models, one for each behavioral mode), (2) infrastructure resources (for example, computers), and (3) attacks. Connecting the first two kinds of models are *conditional probability links*; each such link states how likely a particular behavioral mode of a computational component would be if the infrastructure component that supports the component were in a particular one of its modes (for example, normal or abnormal). We next observe that resources are compromised by attacks that are enabled by vulnerabilities. An attack is capable of compromising a resource in a variety of ways; for example, buffer-overflow attacks are used both to gain control of a specific component and to gain root access to the entire system. However, the variety of compromises enabled by an attack are not equally likely (some are much more difficult than others). We therefore have a third tier in our model describing the ensemble of attacks assumed to be available in the environment, and we connect the attack layer to the resource layer with conditional probability links that state the likelihood of each mode of the compromised resource once the attack is successful. The attack plans generated by computational vulnerability analysis constitute this third tier. However, a transformation is required for them to fulfill this role. Attack plans are and-or trees. However, it is possible (and in fact likely) that different attack plans share subplans (for example, lots of multistage attacks begin with a buffer-overflow attack being used to gain root privilege). Therefore, all the attack plans are merged into a single and-or tree, which constitutes the third tier of the model. The top-level nodes of this tree, which model the desirable properties of the computational resources, are then connected to the second tier (the resource layer) of the model.

We next briefly describe how the diagnostic and monitoring processes use attack plans.

## Diagnostic Reasoning

Diagnosis is initiated when a discrepancy is detected between the expected and actual behaviors of a computation. We use techniques sim-

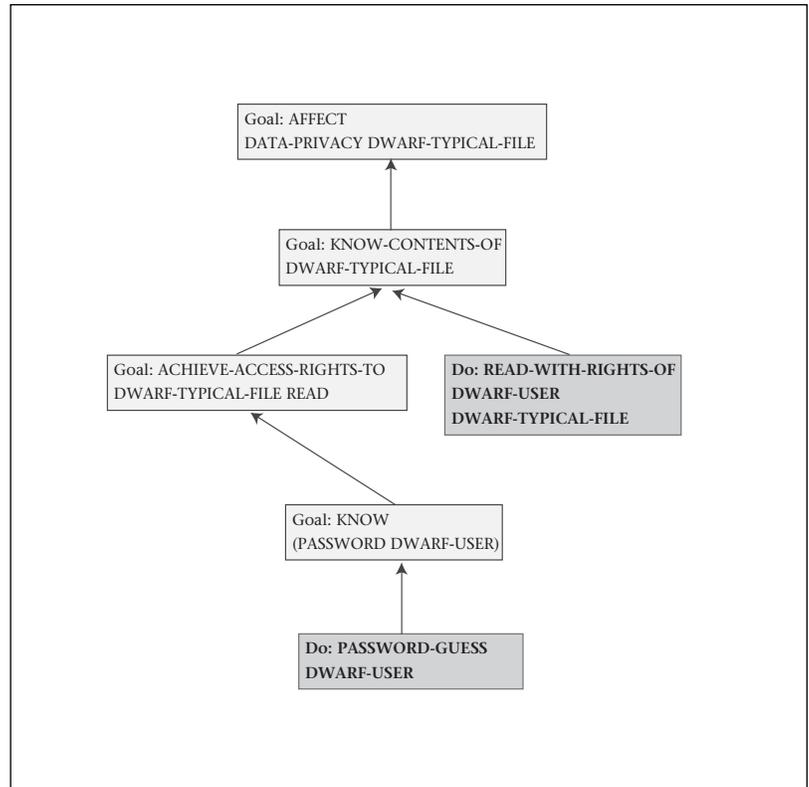


Figure 6. A Second Plan for Affecting Privacy.

ilar to those identified in deKleer and Williams (1989) and Srinivas (1995). We first identify all conflict sets (a choice of behavior modes for each of the computational components that leads to a contradiction) and then proceed to calculate the posterior probabilities of the modes of each of the components. Conflicts are detected by choosing a behavioral mode for each computational component and then running each of the selected behavioral models. If this choice of behavioral mode leads to a contradiction, then the choice of models is a conflict set; otherwise, it is a consistent diagnosis.

Whenever the reasoning process discovers a conflict, it uses dependency tracing (that is, its truth maintenance system) to find the subset of the models in the conflict set that actually contributed to the discrepancy. At this point, a new node is added to the Bayesian network representing the conflict. This node has an incoming arc from every node that participates in the conflict. It has a conditional probability table corresponding to a pure logical and. That is, its true state has a probability of 1.0 if all the incoming nodes are in their true states; otherwise, it has probability 1.0 of being in its false state. Because this node represents a logical

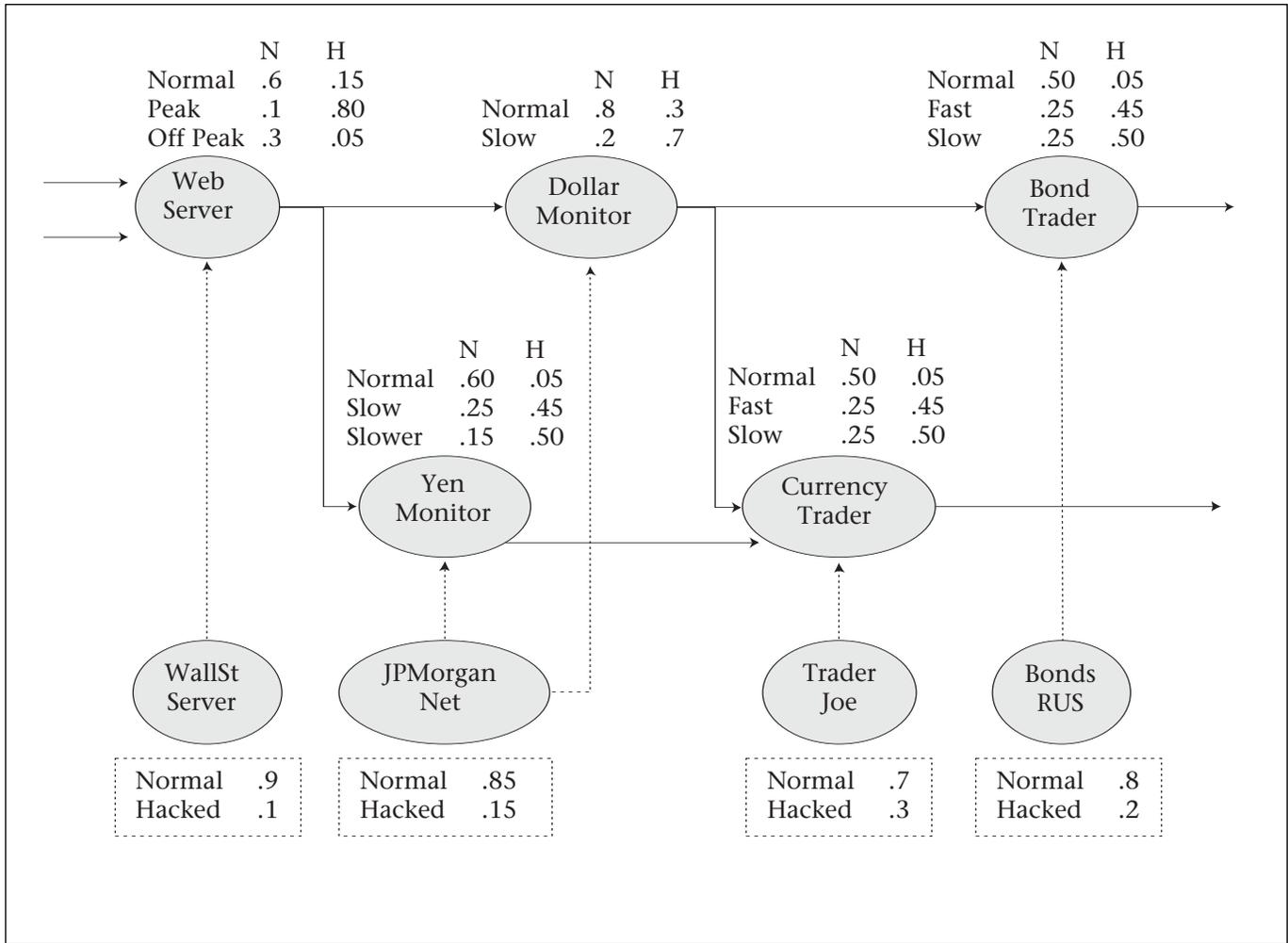


Figure 7. An Example of the Extended System Modeling.

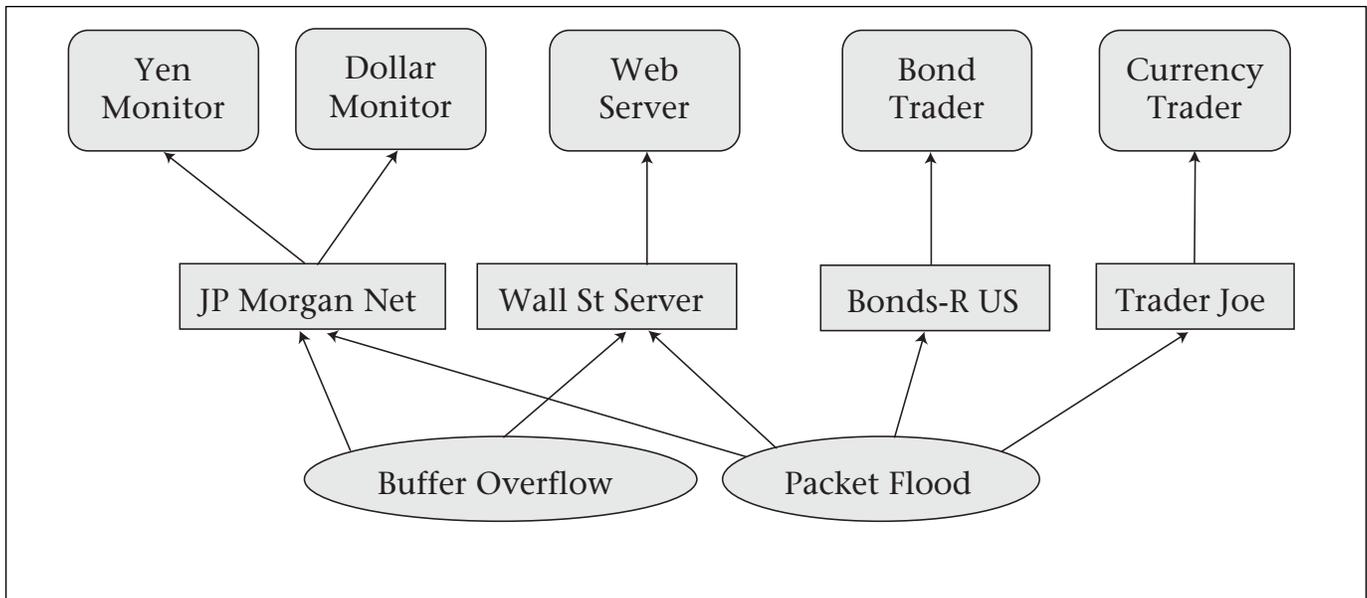


Figure 8. An Example of the Three-Tiered System Modeling Framework.



run through several analyses, including both those in which the output are within the expected range and those in which the output are unexpected. Figure 9 shows the results of the analysis. There are four runs for each case, each with a different attack model developed by computational vulnerability analysis. In the first, there are no attacks present, and the a priori values are used for the probabilities of the different modes of each resource. The second run takes place in an environment in which only a buffer-overflow attack is possible; the third run includes only a packet-flood attack. The fourth run is in an environment in which both types of attacks are possible. Note that the posterior probabilities are different in each case because each set of attack models couples the resource models in a unique manner. These posterior probabilities can then be used to update the overall trust model because each run provides some evidence about compromises to the resources involved. Furthermore, it is possible that a successful attack would have affected additional resources that were not used in the computation being diagnosed; this suspicion is propagated by the Bayesian network. In effect, the reasoning is that the failure of the computation is evidence that a resource has been compromised, which, in turn, is evidence that an attack has succeeded. However, if the attack has succeeded, then other resources sharing the vulnerability might also have been compromised and should be trusted somewhat less in the future.

### Application to Long-Term Monitoring

The long-term monitoring system accepts input from intrusion detectors, firewalls, system logs, and self-diagnostic application systems and attempts to recognize multistage concerted attacks that would otherwise escape attention. Skillful attackers move slowly, first scoping out the structure and weaknesses of a computational environment, then slowly gaining access to resources. Often the process is staged: Access to one resource is used to gain more information about the environment and more access to other resources within it. Computational vulnerability analysis produces attack plans very much like those developed by such skillful attackers (in particular, *red teamers*, who simulate attackers as part of exercises, report thought processes very similar to those developed by our tool).

The monitoring system performs many low-level filtering, collating, and conditioning functions on the data. Once these operations

have been performed, the system attempts to match the data streams to a *trend template*, a model of how a process evolves over time. A trend template is broken along one dimension into data segments, each representing a particular input or the product of applying some filter (that is, smoothing, derivate) to some other data segment. On another dimension, the template is broken into temporal intervals with landmark points separating them. There are constraints linking the data values within the segments (for example, during this period, disk consumption on system 1 is growing rapidly while network traffic is stable). There are also constraints on the length of time in each interval and on the relative placement of the landmark points (for example, the period of disk consumption must be between three days and two weeks; the start of disk consumption must follow the start of network traffic growth).

Trend template recognition is a difficult process. It involves making (usually multiple) assumptions about where each interval begins and then tracking the data as they arrive to determine which hypothesis best matches the data. Within each interval, regression analysis is used to determine degree of fit to the hypothesis. More details are provided in Doyle et al. (2001b).

One source of trend templates is computational vulnerability analysis. Each attack plan actually constitutes a set of trend templates because the attack plans are developed as and-or trees. In contrast to the diagnostic application where the plans are merged, here we unfold each individual attack plan into a set of individual plans by removing the or nodes. Each unfolded plan, therefore, consists of a goal node supported by a single plan node, which, in turn, is supported by a set of goal nodes, all of which must be satisfied for the plan to succeed (these goal nodes are, in turn, supported by individual plan nodes; the recursion continues until terminated by a primitive action node). This tree represents a set of constraints on the temporal ordering: A goal is achieved after all the steps in the plan are achieved, but the plan steps might happen in parallel. Each step is characterized by expectations on the various data streams; we are currently developing the mappings between the attack plan steps and features of data streams that would be indicative of the plan step.

At any point in time, the trend template matcher has an estimate for how well each template matches the data. These estimates are evidence that specific attacks have been launched against specific resources and are

therefore also evidence about the degree and type of compromise present in each resource. Thus, this process, too, contributes to the overall trust model.

## Conclusions and Future Work

I showed how computational vulnerability analysis can model an attack scenario and how such a model can drive both long-term monitoring and diagnostic processes that extract maximum information from the available data. In the case of diagnosis, this means carefully analyzing how unexpected behavior might have arisen from compromises to the resources used in the computation. For long-term monitoring, this means recognizing the signs of a multistage attack by collating evidence from many sources. Both processes contribute to an overall trust model.

The purpose of the trust model is to aid in recovering from a failure and help avoid compromised resources in the future. The trust model functions at the levels of (1) observable behavior, (2) the compromises to the underlying computational resources, and (3) the vulnerabilities and the attacks that exploit them.

Computational vulnerability analysis is an important part of this process. However, it has value beyond its contribution to self-adaptivity. Vulnerability assessments are a useful tool for system administrators as they attempt to keep their environments functioning. Often, such an assessment can spot problems that can be corrected easily, for example, by changing filtering rules or adding a firewall. We have begun to use the tool in our own lab for such assessments and hope to use it more systematically as the coverage grows.

Computational vulnerability analysis can also be a valuable adjunct to intrusion detection systems, helping to collate events over a longer period into systematic attack plans. We have already begun to use this tool in a limited way in our lab to examine and prevent vulnerabilities in various subspaces. We are planning to add more expertise to the system and use it more widely in the future. We are also planning to integrate this tool with the lab's intrusion-detection system.

## 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 Technology Office of the Defense Advanced Research Projects Agency (DARPA) under Space and Naval Warfare Systems Center—San Diego

(SPAWAR) contract N66001-00-C-8078. The views presented are those of the author alone and do not represent the view of DARPA or SPAWAR.

## References

- Davis, R., and Shrobe, H. 1982. Diagnosis Based on Structure and Function. In Proceedings of the Second National Conference on Artificial Intelligence, 137–142. Menlo Park, Calif.: American Association for Artificial Intelligence.
- deKleer, J., and Williams, B. 1989. Diagnosis with Behavior Modes. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 1324–1330. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.
- deKleer, J., and Williams, B. 1987. Diagnosing Multiple Faults. *Artificial Intelligence* 32(1): 97–130.
- Doyle, J.; Kohone, I.; Long, W.; Shrobe, H.; and Szolovits, P. 2001a. Agile Monitoring for Cyberdefense. In Proceedings of the Second DARPA Information Security Conference and Exhibition (DISCEX-II), 318–328. Washington, D.C.: 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, 17–23. Washington, D.C.: IEEE Computer Society.
- Hamscher, W., and Davis, R. 1988. Model-Based Reasoning: Troubleshooting. In *Exploring Artificial Intelligence*, ed. H. Shrobe, 297–346. Menlo Park, Calif.: AAAI Press.
- Shrobe, H. 2001. Model-Based Diagnosis for Information Survivability. In *Self-Adaptive Software*, eds. R. Laddaga, P. Robertson, and H. Shrobe. New York: Springer-Verlag.
- Srinivas, S. 1995. Modeling Techniques and Algorithms for Probabilistic Model-Based Diagnosis and Repair. Technical Report, STAN-CS-TR-95-1553, Stanford University.



**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.

*Call for Proposals*

# Intelligent Systems Demonstrations

The AAI Intelligent Systems Demonstrations program (collocated this year with IJCAI-03) showcases state-of-the-art AI implementations and provides AI researchers with an opportunity to show their research in action. Implemented intelligent systems allow us not only to experimentally validate AI research, but also to make AI research accessible to each other, to the broader scientific community, and to the public at large.

Researchers from all areas of AI are encouraged to submit proposals to demonstrate their systems. Submissions will be evaluated on the basis of their innovation, relevance, scientific contribution, presentation, and “user friendliness,” as well as potential logistical constraints. This program is primarily to encourage the early exhibition of research prototypes, but interesting mature systems and commercial products are also eligible (commercial sales and marketing activities are not appropriate in the Intelligent Systems Demonstration program, and should be arranged as part of the IJCAI-03 Exhibits program). Demonstrations that can be used by the audience and/or that interact with the audience are particularly encouraged.

Demonstration systems should be available as much as possible during the conference exhibition. Each demonstration will have a scheduled and advertised time during which it is the “featured” demonstration. Each accepted demonstration system must be attended by at least one knowledgeable representative (preferably an architect of the system) who will be available to answer in-depth technical questions at scheduled times.

Demonstration proposals must be made electronically using the forms at: [www.cs.rochester.edu/research/ijcai2003/isd/](http://www.cs.rochester.edu/research/ijcai2003/isd/). Researchers who cannot access the world wide web may contact the organizers to make alternative arrangements. In addition to contact information, proposals must include the following, all of which may be submitted via the internet:

1. A two-page description in AAI paper format of the technical content of the demo, including credits and references. These descriptions will appear in the conference proceedings, space permitting.
2. A 150-word summary of the demo in plain text. Please include title, demonstrator names, and affiliation(s). This summary will be used to compile a program for the demonstrations. Please try to keep the descriptions under the 150-word limit.
3. An demo storyboard of not more than six pages total or an informal videotape of the demo (in NTSC VHS format), that describes how the demonstration will proceed (as opposed to the technical merits of the research being demonstrated). This is the committee's primary method of evaluating your proposal. Please emphasize the elements that make your demonstration exciting and interesting. Videotapes (three copies) should be mailed to the address given on the web page.
4. A detailed description of hardware and software requirements. Demonstrators are encouraged to be flexible in their requirements (possibly with different demos for different logistical situations). Please state what you can bring yourself and what you absolutely must have provided. Generally speaking, we can provide generic PCs with standard software such as web browsers, computer monitors, and peripherals such as TVs and VCRs. Each demonstration will be assigned a booth in the Exhibit Hall.

Demo proposals must be received in their entirety including any supporting materials by Friday, February 21, 2003. Authors will be notified of acceptance by March 18, 2003.

We especially hope that authors of papers accepted for presentation at the conference technical program will be able to demonstrate their research in the AAI Intelligent Systems Demonstration Program. To present a system demonstration, however, the authors must still submit a proposal conforming to the above requirements by the Demonstration program deadline. Submitters who wish to demonstrate intelligent mechanical systems that interact with the real world (aka “robots”) should direct their efforts toward the Robot Exhibition.

If you have any questions or comments about the AAI Intelligent Systems Demonstration program, we encourage you to address them to the program organizer, George Ferguson ([ferguson@cs.rochester.edu](mailto:ferguson@cs.rochester.edu)).

**Collocated with the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)**

Acapulco, Mexico ■ August 9-15, 2003