Computational Vulnerability Analysis for Information Survivability *

Howard Shrobe
NE43-839
Artificial Intelligence Laboratory
Massacusetts Institute of Technology
Cambridge, MA 02139
hes@ai.mit.edu

Abstract

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 paper is set in the context of self-adaptive survivable systems: software that judges the trustworthiness of the computational resources in its environment and which 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, fire-walls and all the self-adaptive components, building a composite trust-model used by each component. Self-adaptive surviable systems contain models of their intended behavior, models of the required computational resources, models of the ways in which these resources may be compromised and finally, models of the ways in which a system may be attacked and how such attacks can lead to compromises of the computational resources.

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

*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 Technology Office of the Defense Advanced Research Projects Agency (DARPA) under Space and Naval Warfare Systems Center - San Diego Contract Number N66001-00-C-8078. The views presented are those of the author alone and do not represent the view of DARPA or SPAWAR.

Background and Motivation

The infrastructure of modern society is controlled by computational systems that are vulnerabile 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 to enable it to continue functioning even when attacks have been successfully launched.

Our presmise 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 that method which 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, therfore, 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 priviledge 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" 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. This was followed by a period in which nothing anomalous happened. But then we began to notice that the consumption of disk space and the amount of network traffic from outside the lab were increasing and this continued for some time. Then the load 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 very common *attack plan*. Such attack plans have multiple stages, temporal constraints between the

stages, and constraints within each stage on values and their derivatives (e.g. the rate of growth of disk space consumption). These can, thefore, be used as "trend templates" for collating and analyzing the alerts from intrusion detection systems and the data in common system logs. This 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 paper 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 (e.g. the tell-tales of a password scan) or by *anomaly profiling* (e.g. detecting a difference in behavior between the current process and a statistical norm). Neither of these alone is capable of dealing with a skillful attacker who would stage his 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 in self-diagnosing adaptive systems leads to increased precision in trust modeling and greater survivability of critical systems.

Contributions of this Work

We develop 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 multi-stage attacks. These are crucial issues when failure is caused by a concerted attack by a malicious opponent who is attempting to avoid detection.

We develop a unified framework for reasoning about the failures of computations, about how these failures are related to compromises of the underlying resources, about the vulnerabilities of these resources and how these vulnerabilities enable attacks. We then extend previous work in Model-Based diagnosis (Davis & Shrobe 1982; deKleer & Williams 1987; 1989; Hamscher & Davis 1988; Srinivas 1995) to enable systems capable of self-diagnosis, recovery and adaptation. We also use this framework in building 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 to identify weak links in a computational environment. This can help system administrators improve the security and robustness of their network, often by instituting simple changes. We are 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 of connecting the monitoring system to a commercial instrusion detector. We plan to begin deploying this monitoring system within the next six months.

This paper 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. We conclude by explaing how attack plans fit within the self-diagnostic and the long-term monitoring frameworks.

Computational Vulnerability Analysis

In this section we examine the core issue of this paper which is how to make the modeling of attacks and vulnerabilities systematic.

We do this by grounding the analysis in a comprehensive ontology that covers:

- System properties
- System Types
- System structure
- 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 (e.g. an operating system has a scheduler, a file system, etc.), 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, predicatable 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 (e.g. 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 priviledges). The output of this reasoning is a set of multi-stage attacks, each of which is capable of affecting the property of interest.

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

- 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 sub-systems
  - What type of operating system is in each node
  - How is the operating system decomposed into subsystems
- Server and user software suites: What software functionality is deployed on each node.
- What are the access rights to data and how are they controlled
- What are the places in which data is stored or transmitted

The next step is to model dependencies. To do this we 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
- Privacy of Stored Data

Within the diagnostic framewok 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 while data privacy is a property contributed by the access-control mechanisms.

Control Relationships

We now turn our attention to a rule-base that uses this ontology to reason about how one might affect a desirable property. Our goal is to make this rule base as abstract and general as possible. For example, one such abstract rule says (this is a paraphrase of the actual rule, which is coded in a Lisp-based rule system):

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

This 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. The following are examples of such general and abstract rules:

```
If the goal is to control a compo-
nent ?x
Then find an input ?y to ?x
 and find a way to modify ?y

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

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

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.
- Buffer overflow attacks are capable of taking control of the components that are vulnerable.

One of the rules shown above indicates that one can control a component by modifying its inputs. The following rules describe how this can be done:

```
If the goal is to modify an input ?x of
    component ?y
then find a component ?z which control
     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 in-
put ?x
 and find a way to modify ?z
```

Access Rights

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

- For each type of object we enumerate the *operations* that can be performed on objects of that type.
- For each operation we specify the *capabilities* that are required to perform the operation

- The capabilities are related by a subsumption relationship that forms a DAG.
- For each agent (i.e. a user or a process) we enumerate the capabilities that the agent possesses at any time.
- 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.
- Typically groups of machines manages access rights collectively (e.g. workgroups in MS Windows, NIS in Unix environments). We refer to such a collection of machines as an *access pool*.
- The structure of access pools may be orthogonal to the network topology. Machines in different subnets may be parts of the same access pool, while machines on a common subnet may be members of different access pools.

Given this framework we provide rules that describe how to gain access to objects:

```
If the goal is to gain access to opera-
tion ?x
   on object ?y
and operation ?x on ?y requires capil-
ity ?z
then find a process ?p whose capabil-
ity ?w
     subsumes ?z
 and find a way to take control of ?p.

If the goal is to gain access to opera-
tion ?x
   on object ?y
and operation ?x on ?y requires capil-
ity ?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 capa-
bilty ?w
```

Knowledge of Secrets

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

- To obtain knowledge of a password, find it by guessing, using a guessing attack.
- 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 datam
  - To sniff a piece of data find a file containing the data and gain access to it.
- To obtain knowledge of a password, gain write access to the password file and change it.

Network Structure

The next section of rules deal with networks. As mentioned above, networks are described in terms of the decomposition into subnets and the connections of subnets by routers and switches. In addition, for each subnet we 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 (e.g. 10, 100, and 1000 base-T type ethernet); in these network 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; this 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 net-
work 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 is transformed between various representations. The following issues are modeled:

- Data elements reside in many places
- Executable code resides in many place
  - Main memory
  - Boot files
  - Paging files
- Data elements and code move between their various residences
  - Data migrations go through peripheral controllers
  - Data migrations go thorugh networks

Given these representations we then provide the following rules:

- To modify or observe a data element find a residence of the element and find a way to modify or observe it in that residence.
- 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 of a network segment so as to modify data during transmission. For example:

- To control traffic on a network segment launch "A man in the middle attack" by gaining control of a machine on the network and then finding a way to redirect traffic to that machine rather than to the router or switch.

- To observe network traffic get control of a switch or router and a user machine and the reflect the traffic to the user machine.
- To modify network traffic, launch an ”inserted packet” attack. To do this get control of machine on the network and then send a packet from that machine with the correct serial number 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, linked executable images. In many systems there are other representations as well (e.g. JAR files for Java code). In addition, processes such as compilation and linking transform code between these formats. This leads to the following rules:

- To modify a software component find an upstream representation of that component and then find a way to modify that representation and to cause the transformation between representations to happen.
- To modify a software component gain control of the processes that perform the transformation from upstream to downstrem representation.

An Example

The following example illustrate 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:

- One goal is to control the scheduler of the computer because the scheduler is a component that impacts performance.
- One way to do that is to modify the scheduler’s policy parameters because the policy parameters are inputs to the scheduler.
- One way to do that is by gaining root access to the computer because root access is required to modify these parameters.
- One way to do that 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 apriori* 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 etc. The attack plans include only those which satisfy all these constraints. A computer may be vulnerable to an exploit but if there is a fire-wall isolating it from the attacker, the analysis will not develop an attack plan exploiting that vulnerability.

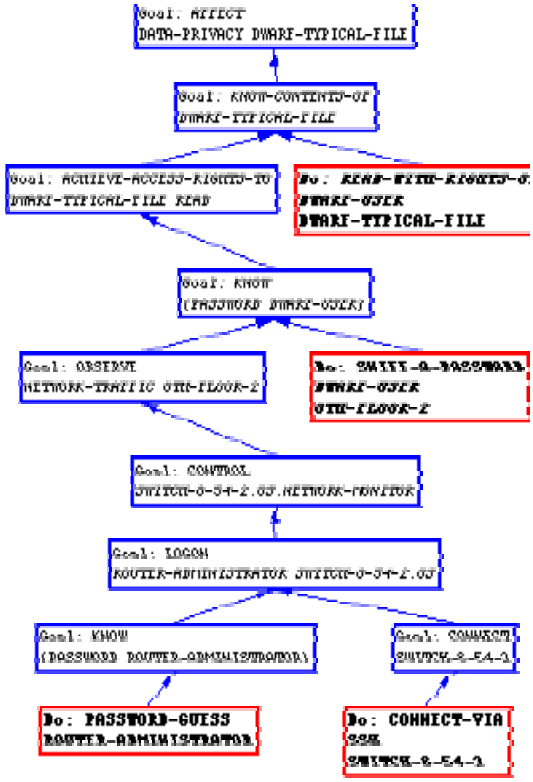

Figure 1: A Plan for Affecting Privacy

Figures 1 and 2 show 2 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 may have several incoming links from Plan nodes; all that is required is that one of the plans work. Plan nodes are And nodes; each sub-goal must be fulfilled for a plan to be valid) The leaves of the tree are primitive actions, i.e. actual attack steps. The figures show one slice through the and-or tree for simplicity.

For the given system description our vulnerability analyzer generated 7 attack plans for the privacy property and 9 plans for attacking performance.

We now turn briefly to the question of how the attack plans are utilitzed in diagnostic reasoning and in long term

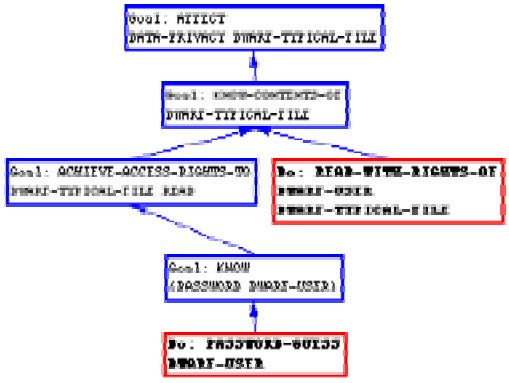

Figure 2: A Second Plan for Affecting Privacy

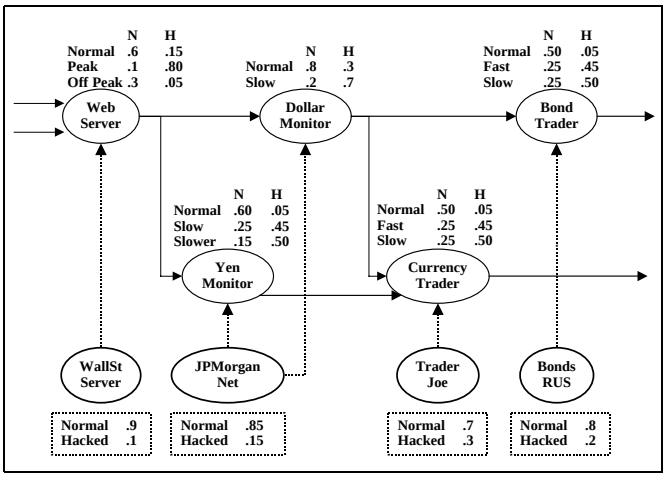

Figure 3: An Example of the Extended System Modeling Framework

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 inputs from intrusion detectors, fire-walls, and self-monitoring applications in an attempt to detect more pernicious, multistage attacks.

We will briefly describe each use in the next two sections.

Application to Diagnosis

Figure 3 shows a model of a fictitious distributed financial system which we use to illustrate the reasoning process. The system consists of five interconnected software modules (Web-server, Dollar-Monitor, Bond-Trader, Yen-Monitor, Currency-Trader) utilizing four underlying computational resources (i.e. the computers WallSt-Server, JPMorgan, BondRUs, 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 4. 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, buffer-overflow and packet-flood. Packet-floods can affect each of the resources because they are all networked systems; buffer-overflows affect only the 2 resources which 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 is an example of a *common mode failure*; intuitively, a common mode failure occurs when a single fault (e.g. an inaccurate power supply), leads to faults at several observable points in the systems (e.g. 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: computational components (represented by a set of input-output relationships and delay models one for each behavioral mode), infrastructural resources (e.g. computers) and 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 infrastructural component that supports that component were in a particular one of its modes (e.g. 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. But 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 likelikhood of each mode of the compromised resource once the attack has been 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 sub-plans (e.g. lots of multi-stage attacks begin with a buffer-overflow attack being used to gain root priviledge). 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 will next briefly describe how the diagnostic and monitoring processes use attack plans.

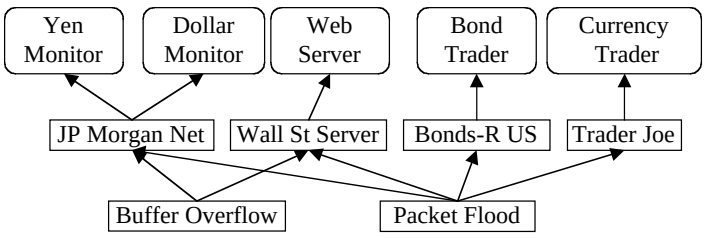

Figure 4: An Example of the Three Tiered System Modeling Framework

Diagnostic Reasoning

Diagnosis is initiated when a discrepancy is detected between the expected and actual behaviors of a computation. We use techniques similar to (deKleer & Williams 1989; 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 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 use dependency tracing (i.e. 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" i.e. its true state has a probability of 1.0 if all the incoming nodes are in their true states and it otherwise has probability 1.0 of being in its false state. Since this node represents a logical contradiction, it is pinned in its false state.

We continue until all possible minimal conflicts are discovered, extending the Bayesian network with a new node for each. At this point any remaining set of behavioral models is a consistent diagnosis; we choose the minimal such sets (i.e we discard any diagnosis that is a superset of some other diagnosis). For each of these we create a node in the Bayesian network which is the logical-and of the nodes corresponding to the behavioral modes of the components. This node represents the probability of this particular diagnosis. The Bayesian network is then solved giving us updated probabilities.

The sample system shown in Figure 3 was run through several analyses including both those in which the outputs are within the expected range and those in which the outputs are unexpected. Figure 5 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 probabities are different in each case. This is because each set of attack models couples the resource models in a unique manner. These posterior probabilities may be then used to update the overall trust model, as each run provides some evidence about compromises to the resources involved. Futhermore, 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: the failure of the computation is evidence that a resource has been compromised; this, in turn, is evidence that an attack has succeeded. But 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.

Slow Fault on both outputs 25 "Diagnoses"
34 Minimal Conflicts
Output of Bond-Trader Observed at 35
Output of Current-trader Observed at 45

Name	Prior	Posterior			
Wallst	.1	.27	.58	.75	.80
JPMorgan	.15	.45	.62	.74	.81
Bonds-R-Us	.20	21	.20	.61	.50
Trader-Joe	.30	.32	.31	.62	.50

Computations Using Each Resource					
Web-Server	Off-Peak	.03	.02	.02	.02
	Peak	.54	.70	.78	.80
	Normal	.43	.28	.20	.18
Dollar-Monitor	Slow	.74	.76	.73	.76
	Normal	.26	.24	.27	.24
Yen-Monitor	Really-Slow	.52	.54	.56	.58
	Slow	.34	.35	.34	.34
	Normal	.14	.11	.10	.08
Bond-Trader	Slow	.59	.57	.76	.70
	Fast	0	0	0	0
	Normal	.41	.43	.24	.30
Currency-Trader	Slow	.61	.54	.62	.56
	Fast	.07	.11	.16	.16
	Normal	.32	.35	.22	.28

Attack Types		Attacks Possible			
Name	Prior	None	.Buffer-Overflow	Packet-Flood	Both
Buffer-Overflow	.4	0	.82	0	.58
Packet-Flood	.5	0	0	.89	.73

Figure 5: Updated Probabilities

Application to Long Term Monitoring

The long term monitoring system accepts inputs from intrusion-detectors, fire-walls, system logs, and self-diagnostic application systems and attempts to recognize multi-stage 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" people 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 (i.e 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 (e.g. 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 (e.g. the period of disk consumption must be between 3 days and 2 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 it arrives 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; this is 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 may 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

We have shown how Computational Vulnerability Analysis can model an attack scenario, 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 multi-stage 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 to 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 by adding a fire-wall. 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.

References

Davis, R., and Shrobe, H. 1982. Diagnosis based on structure and function. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, 137–142. AAAI.

deKleer, J., and Williams, B. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

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

Doyle, J.; Kohone, I.; Long, W.; Shrobe, H.; and Szolovits, P. 2001a. Event recognition beyond signature and anomaly. In *Proceedings of the Second IEEE Information Assurance Workshop*. IEEE Computer Society.

Doyle, J.; Kohone, I.; Long, W.; Shrobe, H.; and Szolovits, P. 2001b. Agile monitoring for cyber defense. In *Proceedings of the Second Darpa Information Security Conference and Exhibition (DISCEX-II)*. IEEE Computer Society.

Hamscher, W., and Davis, R. 1988. Model-based reasoning: Troubleshooting. In Shrobe, H., ed., *Exploring Artificial Intelligence*. AAAI. 297–346.

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

Srinivas, S. 1995. Modeling techinques and algorithms for probablistic model-based diagnosis and repair. Technical Report STAN-CS-TR-95-1553, Stanford University, Stanford, CA.