Using A Cognitive Architecture to Automate Cyberdefense Reasoning

D. Paul Benjamin
Pace University
benjamin@pace.edu

Partha Pal, Franklin Webber, Paul Rubel, Mike Atigetchi
BBN Technolgies, Inc.
ppal@bbn.com, franklin@eutaxy.net, prubel@bbn.com, matighet@bbn.com

Abstract

The CSISM project is designing and implementing an automated cyberdefense decision-making mechanism with expert-level ability. CSISM interprets alerts and observations and takes defensive actions to try to ensure the survivability of the computing capability of the network. The project goal is a difficult one: to produce expert-level response in realtime with uncertain and incomplete information. Our approach is to emulate human reasoning and learning abilities by using a cognitive architecture to embody the reasoning of human cyberdefense experts. This paper focuses on the cognitive reasoning component of CSISM.

1. Introduction and Background

We are developing an autonomous cognitive agent to improve the survivability of computer networks by detecting and countering attacks that require considerable access to and privilege in the system. This agent monitors events and observations about the behavior of the network, formulates hypotheses about attackers and selects actions to maximize the portion of the network that can continue to support computation.

This project is a successor to the DPASA project [1], which attempted to design and develop a survivable version of a military information management system by organizing the functional components and defense mechanisms in a survivability architecture. The resulting survivable system provided limited support for autonomous defensive response. The approach taken in the DPASA project is shown in Figure 1. Alerts from the system were treated as accusations. Accusations from a host H and accusations about H from others were considered to estimate whether H can be trusted or not, and when a host falls below a threshold, a response was suggested. Although the DPASA system was able to successfully survive 75% of attack runs, human experts had to interpret the events outside of this automated mechanism, and arbitrate defensive response overriding its suggestions.

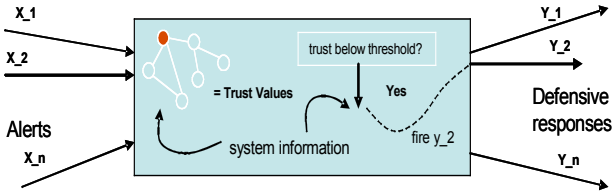

Figure 1: Realization in DPASA

As a follow up to the DPASA effort, we began developing an autonomous cognitive capability that could reason quickly and accurately based on potentially incomplete and inaccurate information contained in reported cyber events. By augmenting survivability architectures (like DPASA) with such a cognitive control loop it will be possible to automate the process of selecting defensive actions in response to cyber attacks. The time constraint for this DARPA program was that the system must provide a response within 250 ms of receiving its initial symptoms, while detecting at least 50% of all attacks with a false positive rate no greater than 10%.

A number of approaches have been used to implement control loops supporting survivability-management decision-making with varying levels of success. But a key part of the control loop, i.e., deciding when to take a response , was based on definitive conditions encoded in the system—either failures or a departure from pre-specified expected behavior.

For instance, the AWDRAT [6] and RMPL [8] approach inserts a cognitive mechanism in parallel with the real execution: a model of the system is embedded in the control loop as shown in Figure 2. The interpretation involves executing the model with the same inputs as the actual system. If system behavior deviates from the model behavior (specified a-priori, by the designers), a response is triggered. The response selection part then considers the utility of potential choices only from the defense's perspective. And there is a learning loop that updates the internal model.

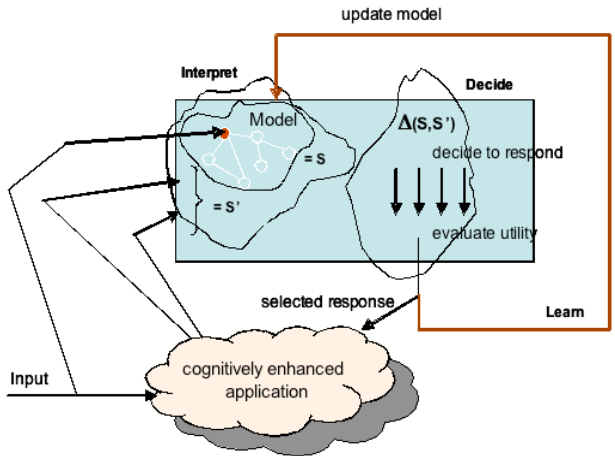

Figure 2: Self-Regenerative Systems Approach

Cortex [4] inserts the decision making process in series with the actual system. In this case, as shown in Figure 3, interpretation involves running the intercepted input though a set of testers. Death of a tester triggers a response. This scheme does not have the option to evaluate possible responses—its first fixed response (an accurate one under the narrowly defined condition that the attack is the input that killed the tester) is to block that specific offending input. It then experiments with the input by varying it along a predefined set of axes and testing them against the tasters trying to come up with a generalized input to block. Even though this mechanism may learn a way resist future repetitions of the same input or its variations, note that it also can only decide and respond to what has been programmed into (i.e., death of a tester) it. Minimizing response time or false positives was not explored actively.

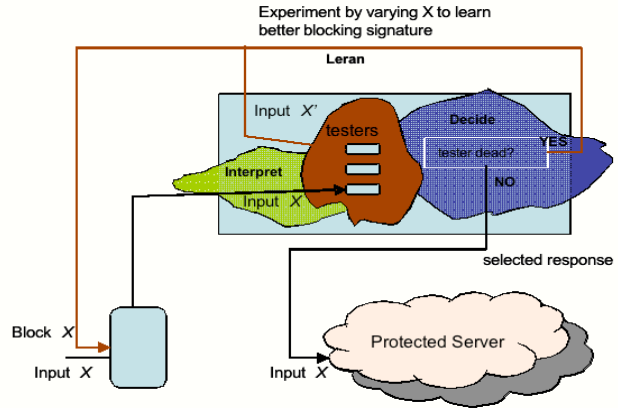

Figure 3: Cortex approach

2. Cognitive Architectures

Advances in cognitive architectures make their application in the context of this effort possible. The field of cognitive science has progressed beyond modeling individual aspects of cognition. In the past twenty years, unified cognitive architectures have been implemented that attempt to model the whole range of human cognitive activity. One of the most notable architectures is Soar [5], which was originally developed at Carnegie-Mellon University. Soar exhibits a wide range of capabilities, including learning to solve problems from experience, concept learning, use of natural language, and the ability to handle complex tasks. The Soar research community has published an extensive literature that demonstrates clearly successful programs of implementing cognitive mechanisms and their application to a wide range of tasks.

Declarative knowledge in Soar resides in its *working memory*, which contains all the facts Soar knows at any instant. Procedural knowledge in Soar is represented as *operators*, which are organized into *problem spaces*. Each operator is implemented by a set of *rules*. Each problem space contains the operators relevant to interacting with some aspect of the system's environment. In our system, some problem spaces contain operators describing how CSISM interprets alerts or analyzes the attacker's goals. Other problem spaces contain operators that contain system-specific knowledge, apply coherence reasoning, and propagate truth values. At each step, Soar must choose one operator from one of these problem spaces to execute.

The basic problem-solving mechanism in Soar is universal subgoaling: every time there is choice of two or more operators, Soar creates a subgoal of deciding which to select, and brings the entire knowledge of the system to bear on solving this subgoal by selecting a problem space and beginning to search. This search can itself encounter situations in which two or more operators can fire, which in turn causes subgoals to be created, etc. When an operator is successfully chosen, the corresponding subgoal has been solved and the entire solution process is summarized in a new rule, called a chunk, which contains the general conditions necessary for that operator to be chosen. This rule is added to the system, so that in similar future situations the search can be avoided. In this way, Soar learns.

A distinctive aspect of our project is the use of such a comprehensive, unified cognitive architecture to create an autonomous cybersecurity agent. Soar is the architecture we have chosen because of its emphasis on problem solving and learning mechanisms. Using Soar enables us to bring these general and powerful abilities to bear on the task of building a cybersecurity agent. In particular, Soar can evaluate possible predictions and responses in a very flexible way. Soar's reasoning and sub-goaling abilities enable it to bring a wide range of knowledge to bear on reasoning about failures, not just a predefined set of patterns, and it can learn new rules incorporating all of this knowledge.

3. Design and Implementation of CSISM

Our system is named CSISM (Cognitive Support for Intelligent Survivability Management), and consists of two main components: a network simulator and the cognitive reasoner. These are connected by Java. The network simulator is written in JESS (Java Expert System Shell), and provides a high-level abstraction of a computer network.

The distributed system we simulate is the survivable version of the information management system developed in the DPASA project. This system was transitioned from BBN development lab into AFRL in 2005 for red team exercises. This network consists of 47 hosts organized into four LANs and four quads. The hosts are heterogeneous, running four different operating systems: SeLinux, Windows XP, Solaris and Windows 2000.

The CSISM project aims to augment defense-enabled systems with a survivability architecture like the DPASA system. Therefore, a first step was to analyze the logs and results of attacks on the DPASA system to understand what the sensors embedded in the architecture observed, reported, and how they were connected to defensive responses by the human defenders. This analysis identified five types of expert knowledge that needed to be implemented:

Symptomatic knowledge, which consists of possible explanations for anomalous events, e.g. "if A says B is unresponsive, then either A is lying and therefore corrupt, B is dead, or some intermediate network component is bad."

Reactive knowledge, which consists of possible countermeasures to an attack, e.g. "if A is corrupt and has been rebooted since that was known, consider quarantining A".

Teleological knowledge, which consists of possible attacker goals in a given situation, e.g. "if P and Q belong to a Byzantine fault-tolerant group G with 6 or fewer members, and both P and Q have accused non-group elements of corruption, then corruption of G is a likely goal".

Malicious knowledge, which consists of possible attacker goals in a given situation, e.g. "if the goal is corruption of group G, consider attacks against members of G".

Relational knowledge, which consists of constraints that reinforce or eliminate possible explanations, e.g. "A is a JBI client process on the environmental LAN".

The difficulty that confronted us in designing the reasoning component was that we needed a reasoning component that could perform these types of reasoning used by the human experts, yet do so in a very efficient way to reach a conclusion in less than 250ms. This meant that we needed mechanisms for reasoning deductively to derive conclusions that the human experts derived, and mechanisms for handling a large amount of data, including incomplete and inaccurate data. We could not use existing systems for these purposes, e.g. a theorem prover, because they were too slow.

Our solution was to implement a fast deductive model checker and integrate it with a coherence reasoner in Soar.

Coherence theory [7] is a technique of partial constraint satisfaction [2,3] that searches to find the assignment of hypotheses (either "Accepted" or "Rejected") that is most consistent with the observations. Coherence theory has been applied to a number of tasks, and in particular has been successfully used to model human scientific reasoning [7]. We chose coherence theory because of its psychological plausibility and also because it can be implemented very efficiently. Thagard [7] gives results of several implementations, and found that a greedy algorithm performed as well as more comprehensive algorithms.

We integrate coherence and deduction by using a single representation for the system's beliefs, and performing both types of reasoning on this representation. This is a novel approach to integrating deduction and optimization.

We accomplish this by modeling beliefs about the behavior of the computer network as nodes in a graph. These "hypothesis" nodes are connected by constraint links that embody relationships between the hypotheses. These links can be deductive or coherence links.

The greedy coherence algorithm is a simple hill climber. It first computes a score for each hypothesis node that is the total of the weights of its satisfied constraints minus its unsatisfied constraints. (A positive constraint is satisfied iff both its endpoints are in the same set; a negative constraint is satisfied iff its endpoints are in different sets.) Moving a node with a negative score to the other set will increase the overall score of the hypothesis network (the sum of all the scores of the nodes.) The greedy algorithm moves the node with the highest negative score to the other set, then repeats the whole process until there are no nodes with negative scores.

The reasoner's goal is to find an assignment of hypothesis nodes to "Accepted" or "Rejected" that maximally satisfies the constraints while simultaneously respecting the deductive conclusions. CSISM first performs deductive inference to identify hypotheses that are definitely true or false. These are the "islands of certainty" in the hypothesis graph. The reasoner then searches for an assignment of "Accepted" and "Rejected" to the remaining nodes that maximizes the satisfaction of constraints. CSISM then takes defensive action against hosts identified by Accepted hypotheses with high coherence scores.

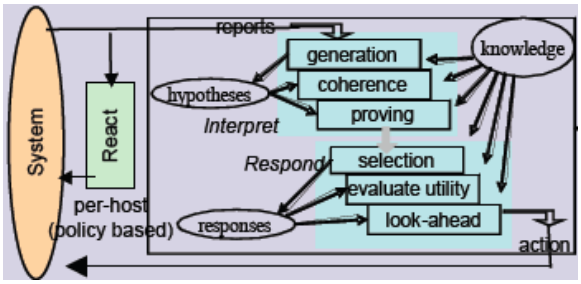

Figure 4. Overall CSISM structure.

Let's consider an example. Three CSISM rules are:

"if host x accuses y of an Omission failure, then either x is corrupt or y is dead or the communications between x and y are broken"
accuses(x,y,Omission) ->
corrupt(x) | dead(y) | comBroken(x,y)

"if a host is dead then it is corrupt."
corrupt(x) -> dead(x)

"if host x accuses host y then host x is not dead."
accuses(x,y) -> -dead(x)

When host_72 accuses host_40, CSISM uses these rules to create a small graph of nodes:

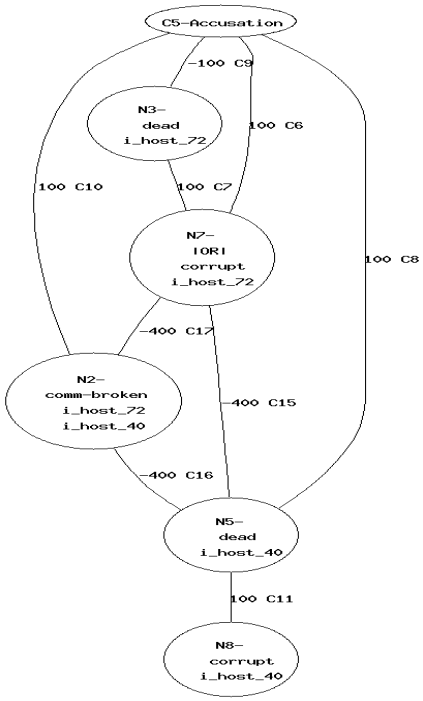

Figure 5. Graph of nodes resulting from an accusation.

As more alerts and observations arrive, CSISM grows this structure into a graph of hypotheses and constraints, which can contain more than a hundred nodes and a few hundred constraints. Constraints may reflect either deductive relationships or coherence relationships, which are co-occurrence relationships. The events of DPASA Run 11 (which occurred in November, 2005) lead to the following graph (which has two separate components):

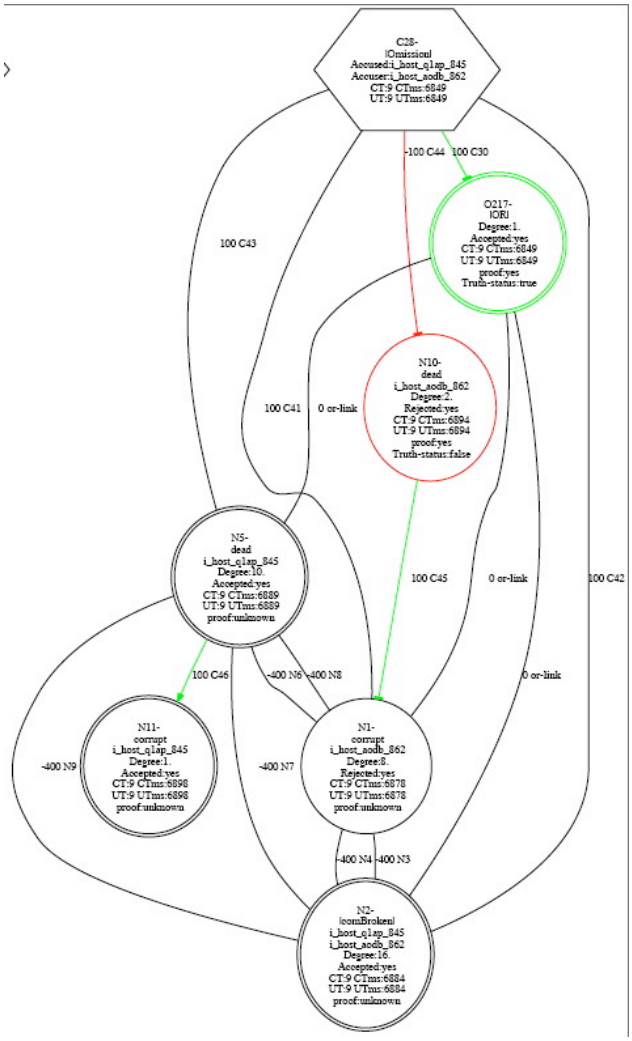

Figure 6a. First component of hypothesis graph in Run 11.

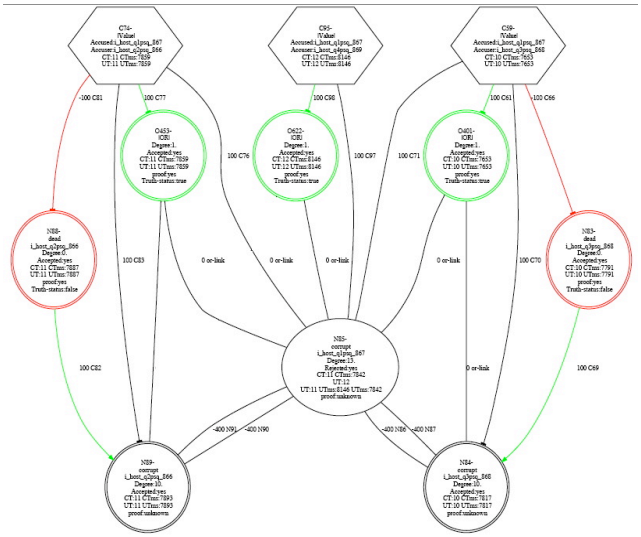

Figure 6b. Second component of hypothesis graph in Run 11.

In these two components, we see nodes (the hexagons are inputs), and links reflecting deductive and coherence relationships. Nodes contain relevant information, including the type of hypothesis and the hosts involved. Also included is the truth status of the hypothesis, which can be "proved true", "proved false" or "unproven". For example, in the graph in Figure 6a, the top-most circle's hypothesis is proven true and the hypothesis in the circle below it is proven false. All the others are unprovable at this point. The true hypothesis is labeled "Accepted" and the false hypothesis is labeled "Rejected".

CSISM's model checker works by testing the consistency of cases for an hypothesis. For example, the model checker tries to determine what operating system platform(s) the attacker has exploited by testing each possible platform. It starts by selecting an operating system, e.g. Windows, and attaching an hypothesis to the graph that states that Windows is exploited, and attaching hypotheses to the graph for the other operating systems stating that those platforms are not exploited. The truth values resulting from these hypotheses are propagated throughout the graph. If a contradiction results, then the reasoner concludes that the selected operating system is not the one exploited. If no contradiction results, then the case is consistent, and that platform is a possibility. In this case, any hypotheses proved true or false are added to a list of possible new theorems; any of these possible theorems that hold in all consistent cases are new theorems.

Once the deductive inference has taken place, the coherence reasoning tries to assign "Accepted" and "Rejected" to the nodes in a manner that is consistent with the deductive results, i.e. it tries to extend the deductive results to the unknown nodes.

Links with numeric scores reflect likelihood of co-occurrence of those nodes; a positive number means that those nodes are likely in the same set (both Accepted or both Rejected, so they are cooperating) and a negative link means those nodes are likely in different sets (they are competing). A greedy search is used to reassign nodes to maximize the weighted satisfaction of links.

This example graph reflects the reasoner's knowledge that one host is certainly known not to be dead (host_aodb) and its belief that probably that host is not corrupt at all, but that the problem lies with host_q1ap, which is believed to be dead. CSISM will respond to this interpretation by quarantining or rebooting host_q1ap.

The graph evolves over time. As new observations and alerts arrive, the graph will expand. When CSISM reboots a machine or otherwise resets its status, the symptoms and hypotheses for that machine are removed from the graph.

Currently, CSISM's reasoner comprises about 500 Soar rules. These reason about Omission accusations (failure to act or respond), Policy accusations (violations of stated policy), Value accusations (incorrect data) and Timing accusations and alerts, as well as flooding attacks. The reasoner is capable of deducing when an attacker possesses an exploit that works only on a single platform and when he has multiple platform exploits. In addition, it reasons about the communications connectivity of the network and what it implies for the status of the machines on the network.

CSISM has been built and tested on the scenarios that occurred in the DPASA tests in 2005, and works well on them. Additionally, new tests have been developed and an automatic problem generator has been implemented to create a wide range of new scenarios for evaluating the event interpretation in CSISM. CSISM passes all the tests we have devised.

Testing is continuing, aimed at the approaching Red Team tests scheduled to occur in May 2008. These tests will assess the entire CSISM system, focusing on the accuracy of the interpretive ability of the cognitive reasoner.

4. Future Work

In its current configuration, CSISM does not exploit Soar's speedup learning ability. There is ample opportunity to apply speedup learning throughout the system, including both the deduction and coherence reasoning. Once the Red Team tests have been completed, this will definitely be done.

In addition, although the greedy algorithm for coherence search has performed well, it is likely that as the hypothesis graphs become larger that it will find local maxima. Our planned approach to this problem is to use machine learning methods to learn groups of hypotheses that tend to occur together, and move them as a group when performing coherence search.

CSISM does not fully exploit Soar's lookahead search ability. We intend to apply lookahead search within the simulator to explore different ways in which the computer network's behavior might evolve, and use these results in reasoning.

5. Acknowledgements

This research was funded by DARPA under Navy Contract No. N00178-07-C-2003.

6. References

[1] J. Chong, P. Pal, M. Atighetchi, P. Rubel, and F. Webber. Survivability Architecture of a Mission Critical System: The DPASA Example. 21st Annual Computer Security Applications Conference, Tucson, Arizona, December 2005

[2] Descorte, Y. and Latombe, J.C., Making compromises among antagonistic constraints in a planner, *Artificial Intelligence 27,* 183-217, 1985.

[3] Freuder, Eugene C. and Wallace, Richard J., Partial Constraint Satisfaction, Artificial Intelligence, special issue on constraint-based reasoning, Volume 58, Issue 1-3, pp.21-70, Elsevier, December, 1992.

[4] K. Z. Haigh, C. Geib. CORTEX Presentation at the Self Regenerative Systems PI Meeting December 2005, http://www.tolerantsystems.org/SRSPIMeeting12/12pi_meeting.html

[5] Laird, J.E., Newell, A. and Rosenbloom, P.S., 1987. "Soar: An Architecture for General Intelligence", Artificial Intelligence 33, pp.1-64.

[6] H. Shrobe, B. Balzer. AWDRAT Presentation at the Self Regenerative Systems PI Meeting December 2005. http://www.tolerantsystems.org/SRSPIMeeting12/12pi_meeting.html

[7] Thagard, Paul, Coherence in Thought and Action, MIT Press, 2000.

[8] B. Williams, P. Robertson. Pervasive Self-Regeneration through Concurrent Model-Based Execution. Presentation at the Self Regenerative Systems Kickoff Meeting, http://www.tolerantsystems.org/SRSProgram/srs_program.html, 2004.