Boar: An Autonomous Agent for Network Intrusion Detection Analysis

By

Archana Perumal

Submitted in partial fulfillment
Of the requirements for the degree of
Master of Science in Computer Science

At

School of Computer Science and
Information Systems
Pace University
December 2004
I hereby certify that this thesis, submitted by Archana Perumal, satisfies the thesis requirements for the degree of Master of Science and has been approved.

Dr. D. Paul Benjamin  
Thesis Advisor  
School of Computer Science and Information Systems  
Pace University  

Date
Abstract

Boar: An Autonomous Agent for Network Intrusion Detection Analysis

Boar is an autonomous network security agent for network intrusion detection which is built on the cognitive architecture of Soar. Boar is a part of VMSoar project. VMSoar is a cognitive network security agent designed for both network configuration and long-term security management. VMSoar performs vulnerability assessments by using VMWare to create a virtual copy of the target machine and then attacking the simulated machine with a wide assortment of exploits. These exploits are captured and logged by the Tcpdump utility tool which runs in the background. This data logged by VMSoar is analyzed by Boar to understand the intrusions and the possible actions that can be taken in order to protect the system and network from exploits and intrusions.

Some important features of Boar that have to be noted is,

1. Boar is just a prototype of the advanced cognitive IDS, which has the capability of learning its environment and adapting to it.
2. Boar can control the capture of data packets in real-time. But for this prototype version, the Boar’s feed is from the log file created by VMSoar.
3. Current version of Boar just logs the alerts when some malicious activity is found. The advanced version might include taking decisions and performing actions like shutting down a port or deflecting the attack and so on in real-time.
4. Boar at this stage can detect only some basic intrusions like port scanning (intruder trying to find which ports are open in the target computer and using that port to launch his attack), invalid TCP flag combinations (like SYN and FIN flag cannot be set in the same packet, etc.) and suspicious activity at the prohibited port (say, in our university we are no longer allowed to use the FTP port 21).

Since Boar is a cognitive model, once fully developed it can start adapting to its environment by learning using a special mechanism called chunking, which is the characteristic feature of Soar’s cognitive architecture.
Boar uses both anomaly and signature based techniques to detect the intrusion. Boar in conjunction with VMSoar will implement the in-depth defense strategy against the intruders which includes prevention, preemption, deterrence, deflection and countermeasures against the intrusion and intruders. In today’s world there is no comprehensive tool that implements all these in-depth defense strategies and provides full automation without human interference.

In common terms, since we are talking about cognitive architecture which tries to mimic human behavior, we can compare Boar to that of our eyes and VMSoar to that of our brain. We see what’s happening in front of us, try to describe what we see and our brain calculates the possibility what might happen next and rationalize the sequence of events. If I see an object being thrown at me, I first see what sort of object it is and then I think about the possible scenarios, where; I can catch that object (if it is a pen) or I can get down so that I do not get hurt (if it is a broken glass, or something that can hurt me) or catch the object and throw it back at/to the source (if it is a ball). When intrusion is compared to the object thrown, basic IDS cannot take all these decisions. Without cognitive modeling it is not possible to take all these decisions in real-time; you need human intervention. This is the basis for Boar and VMSoar project.

The choice of why we are using Soar when there are lot of other Rule-based languages and systems and choice of Tcpdump as the packet logger and not some other tools like Snort are described in detail in the following sections of this paper.
# Table Of Contents

- Abstract iii
- Table of contents v
- List of Abbreviations vii
- Introduction
- Intrusion Detection System 1
  - Intrusion detection 1
  - Categories of IDS 1
  - Host-based IDS 2
  - Network-based IDS 3
  - Need for ID 4
  - Threat 5
  - Vulnerabilities 5
  - Anomaly detection 5
    - Self learning systems 6
    - Simple rule-based 6
  - Signature detection 6
    - Simple rule-based 6
    - Expert systems 6
  - False Positives/negatives 7
  - Limitations of current IDS 7
  - Problem with signatures 7
- Rule Based Systems 9
  - Rule-based system components 9
  - Theory of RBS 10
  - Methods of RBS 12
    - Forward chaining 12
    - Backward chaining 13
    - Forward vs. Backward chaining 13
- Soar 14
  - Unified Theory of Cognition 14
  - Soar as an AI tool 15
  - Soar components 16
    - Knowledge 16
    - Goal directed action 16
    - Reaction 16
    - Learning 17
    - Capabilities 17
  - Long-term knowledge 18
  - Production rules 19
  - Execution cycle 19
  - Impasses 20
Simplified version of Soar Algorithm 20
Learning by Chunking 21
Soar and RBS 21
- Parallel, associative memory 22
- Automated belief maintenance 22
- Preference analyzer 22
- Operator 22
- Rules 23
- Automatic subgoaling 23
- Task decomposition 23

➢ Tcpendump 24
  Overview 24
  What tcpendump can do? 24
  Why tcpendump not any other tool? 25
  Limitations of tcpendump 26
  Understanding dump data 26
  TCP output format 27

➢ Boar Development Environment 27
  Boar Framework 28
  Software Integration 28
  - Tcl 28
  - Soar and tcl 31
  - The input/output links and tcl 32
  - Java, tcl and soar 33
  - Feather 33
  - Java and tcpendump 34

➢ Boar 35
  Boar Feed 35
  Port Scan 36
  Invalid TCP flag combinations 37
  Activity on Forbidden port 38

➢ Boar Rules (Sample source code) 38
➢ Future enhancement and scope 39
➢ Bibliography 40
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>ID</td>
<td>Intrusion Detection</td>
</tr>
<tr>
<td>IDS</td>
<td>Intrusion Detection Systems</td>
</tr>
<tr>
<td>IMAP</td>
<td>Internet Message Access Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>KB</td>
<td>Knowledge Base</td>
</tr>
<tr>
<td>RBS</td>
<td>Rule-Based Systems</td>
</tr>
<tr>
<td>TCL</td>
<td>Tool Command Language</td>
</tr>
<tr>
<td>TCP</td>
<td>Transfer Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
</tbody>
</table>
Intrusion Detection
Intrusion Detection (ID) is the art of detecting inappropriate, incorrect, or anomalous activity. It is a security service that monitors and analyzes system events for the purpose of finding, and providing real-time or near real-time warning of, attempts to access system resources in an unauthorized manner. ID systems that operate on a host to detect malicious activity on that host are called **host-based ID systems**, and ID systems that operate on network data flows are called **network-based ID systems**.

Intrusion Detection System
In more common sense an intrusion detection system is an electronic system designed to protect a specific portal, volume or area, using technologies designed to sense movement, sound or a specific act such as opening a door. This security alarm system consists of various types of sensors (vibration, capacitance, volumetric, etc.) to detect the unauthorized intrusion into a facility. Typical systems include ultrasonic, infrared, microwave sensors, and door switches. IDS systems can be local or connected to a central station.

Network Intrusion Detection Systems are the ‘burglar alarms’ or ‘intrusion alarms’ of the computer and network security field. The aim is to defend a system by using a combination of an alarm that sounds whenever the site’s security has been compromised, and an entity – most often a site security officer that can respond to the alarm and take the appropriate action. An IDS tries to identify attempts to hack or break into a computer system or to misuse it. IDSs may monitor packets passing over the network, monitor system files, monitor log files, or set up deception systems that attempt to trap hackers.

Categories of IDS
There are several ways to categorize an IDS:

- **misuse detection** vs. **anomaly detection**: in misuse detection, the IDS analyzes the information it gathers and compares it to large databases of attack signatures. Essentially, the IDS looks for a specific attack that has already been documented. Like a virus detection system, misuse detection software is only as good as the database of
attack signatures that it uses to compare packets against. In anomaly detection, the system administrator defines the baseline, or normal, state of the network’s traffic load, breakdown, protocol, and typical packet size. The anomaly detector monitors network segments to compare their state to the normal baseline and look for anomalies.

- **network-based vs. host-based systems**: in a network-based system, or NIDS, the individual packets flowing through a network are analyzed. The NIDS can detect malicious packets that are designed to be overlooked by a firewall’s simplistic filtering rules. In a host-based system, the IDS examines at the activity on each individual computer or host.

- **passive system vs. reactive system**: in a passive system, the IDS detects a potential security breach, logs the information and signals an alert. In a reactive system, the IDS responds to the suspicious activity by logging off a user or by reprogramming the firewall to block network traffic from the suspected malicious source.

**What is host-based ID?**

Host-based ID involves loading a piece or pieces of software on the system to be monitored. The loaded software uses log files and/or the system's auditing agents as sources of data. In contrast, a network-based ID system monitors the traffic on its network segment as a data source. Both network-based and host-based ID sensors have pros and cons, and in the end, you'll probably want to use a combination of each. The person responsible for monitoring the IDS needs to be an alert, competent System Administrator, who is familiar with the host machine, network connections, users and their habits, and all software installed on the machine. This doesn't mean that he or she must be an expert on the software itself, but rather needs a feel for how the machine is supposed to be running and what programs are legitimate. Many break-ins have been contained by attentive System Administrators who have noticed something "different" about their machines or who have noticed a user logged on at a time atypical for that user.

Host-based ID involves not only looking at the communications traffic in and out of a single computer, but also checking the integrity of your system files and watching for suspicious processes. To get complete coverage at your site with host-based ID, you need to load the ID
software on every computer. There are two primary classes of host-based intrusion detection software: host wrappers/personal firewalls and agent-based software. Either approach is much more effective in detecting trusted-insider attacks (so-called anomalous activity) than is network-based ID, and both are relatively effective for detecting attacks from the outside.

Host wrappers or personal firewalls can be configured to look at all network packets, connection attempts, or login attempts to the monitored machine. This can also include dial-in attempts or other non-network related communication ports. The best known examples of wrapper packages are TCPWrappers Nuke Nabber for Windows. Personal firewalls can also detect software on the host attempting to connect to the network, such as WRQ's AtGuard. In addition, host-based agents may be able to monitor accesses and changes to critical system files and changes in user privilege.

**What is network-based Intrusion detection?**

A network-based ID system monitors the traffic on its network segment as a data source. This is generally accomplished by placing the network interface card in promiscuous mode to capture all network traffic that crosses its network segment. Network traffic on other segments, and traffic on other means of communication (like phone lines) can't be monitored. Both network-based and host-based ID sensors have pros and cons. In the end, you'll probably want a combination of both.

Network-based ID involves looking at the packets on the network as they pass by some sensor. The sensor can only see the packets that happen to be carried on the network segment it’s attached to. Packets are considered to be of interest if they match a signature. Three primary types of signatures are string signatures, port signatures, and header condition signatures.

String signatures look for a text string that indicates a possible attack. An example string signature for UNIX might be "cat "++-" > /.hosts", which if successful, might cause a UNIX system to become extremely vulnerable to network attack. To refine the string signature to reduce the number of false positives, it may be necessary to use a compound string signature. A compound string signature for a common Web server attack might be "cgi-bin" AND
“aglimpse” AND “IFS”.

Port signatures simply watch for connection attempts to well-known, frequently attacked ports. Examples of these ports include telnet (TCP port 23), FTP (TCP port 21/20), SUNRPC (TCP/UDP port 111), and IMAP (TCP port 143). If any of these ports aren’t used by the site, then incoming packets to these ports are suspicious.

Header signatures watch for dangerous or illogical combinations in packet headers. The most famous example is Winnuke, where a packet is destined for a NetBIOS port and the Urgent pointer, or Out Of Band pointer is set. This resulted in the "blue screen of death" for Windows systems. Another well-known header signature is a TCP packet with both the SYN and FIN flags set, signifying that the requestor wishes to start and stop a connection at the same time. A good ID capability will use both host- and network-based systems. Figuring out where to use each type and how to integrate the data is a real and growing concern.

**Truly effective IDS will use a combination of network- and host-based intrusion detection. Figuring out where to use each type and how to integrate the data is a real and growing concern.**

**Examples for commercial IDS:**

- Snort
- AXENT
- Cisco
- CyberSafe
- ISS
- SHadow

**Need for Intrusion Detection:**

Intrusion detection is needed in today’s computing environment because it is impossible to keep pace with the current and potential threats and vulnerabilities in the computing systems. The environment is constantly evolving and changing fueled by new technology and the
Internet. To make matters worse, threats and vulnerabilities in this environment are also constantly evolving. Intrusion detection products are tools to assist in managing threats and vulnerabilities in this changing environment.

**Threats** are people or groups who have the potential to compromise the computer system or the network system. It may be a curious teenager, a disgruntled employee, or espionage from a rival company or a foreign government. The hacker has become a nemesis to many companies.

**Vulnerabilities** are weaknesses in the systems. Vulnerabilities can be exploited and used to compromise the integrity of the system. New vulnerabilities are discovered every day. Every new technology, product, or system brings with it a new generation of bugs and unintended conflicts or flaws. Also, the possible impacts from exploiting these vulnerabilities are constantly evolving. In a worst-case scenario, an intrusion may cause production downtime, sabotage of critical information, theft of confidential information, cash, or other assets, or even negative public relations that may affect a company’s stock price.

Intrusion detection products are tools that can assist in protecting a company from intrusion by expanding the options available to manage the risk from threats and vulnerabilities. Intrusion detection capabilities can help a company secure its information. The tool could be used to detect an intruder, identify and stop the intruder, support investigations to find out how the intruder got in, and stop the exploit from use by future intruders. The correction should be applied across the enterprise to all similar platforms. Intrusion detection products can become a very powerful tool in the information security practitioner’s tool kit.

The most common approaches to Intrusion detection are statistical anomaly detection and pattern-matching detection or Signature detection.

**Anomaly detection and Signature detection**

**Anomaly detection**
In anomaly detection we watch not for known intrusion - the signal but rather for abnormalities in the traffic in question; we take the attitude that something that is abnormal is probably
suspicious. The construction of such a detector starts by forming an opinion on what constitutes normal for the observed subject (which can be a computer system, a particular user etc.), and then deciding on what percentage of the activity to flag as abnormal, and how to make this particular decision. This detection principle thus flags behavior that is unlikely to originate from the normal process, without regard to actual intrusion scenarios.

**Self-learning systems:** Self-learning systems learn by example what constitutes normal for the installation; typically by observing traffic for an extended period of time and building some model of the underlying process.

**Simple rule-based** - here the user provides the system with simple but still compound rules to apply to the collected statistics.

**Signature detection**
In signature detection the intrusion detection decision is formed on the basis of knowledge of a model of the intrusive process and what traces it ought to leave in the observed system. We can define in any and all instances what constitutes legal or illegal behavior, and compare the observed behavior accordingly.

**Expert-system**
An expert system is employed to reason about the security state of the system, given rules that describe intrusive behavior. Often forward-chaining, production-based tool are used, since these are most appropriate when dealing with systems where new facts (audit events) are constantly entered into the system. These expert systems are often of considerable power and flexibility, allowing the user access to powerful mechanisms such as unification. This often comes at a cost to execution speed when compared with simpler methods.

**Simple rule-based**
These systems are similar to the more powerful expert system, but not as advanced. This often leads to speedier execution.
Boar is a combination of Simple Rule based and Complex learning when fully developed.

False Positives and False Negatives:
False Positives: something occurs that causes IDS to incorrectly identify an intrusion when none has occurred.
False Negatives: something occurs that causes IDS to incorrectly fail to identify an intrusion when one has in fact occurred.
Accuracy of IDS: reflect the number of false positives.
Completeness: reflect the number of false negatives.

Limitations of current IDSs

There are several limitations with existing IDS products. First and foremost, current systems stop at detecting the attack – they do little or nothing to stop the attack in real-time. They report alerts to a centralized database and hope that a network administrator is immediately on-hand to take action. By the time the administrator makes sense of the alert reports and commences counteraction, it is often too late – critical systems may have been compromised and vital information may have been stolen. A perfect IDS should be one that provides intrusion detection and real-time prevention capability against attacks, in a fully automated environment.

There are lot of IDS available, why Boar?
The problem with signatures: What Snort and other signature based intrusion detection systems count on is that malicious traffic will have unique patterns to it that can be matched against rules in the database. For example Snort uses the following rule to look for the SubSeven Trojan:
alert tcp $EXTERNAL_NET any -> $HOME_NET 27374 (msg: "BACKDOOR SIG - SubSeven 22"; flags: A+; content: "; reference:arachnids,485;)
alert

The important part of this rule to note is that Snort is looking for the hex signature "0d 0a 5b 52 50 4c 5d 30 30 32 0d 0a" that is located anywhere in the payload of the packet.
It then seems obvious that there are many ways of circumventing this signature. The first thing that we could do is vary the destination port. This is usually undesirable though since the infected machine is probably using the default port for SubSeven to make it easier to scan for. If the attacker knows what port SubSeven should be running on then they could quickly and easily scan large blocks of addresses for machines listening on that port. The next evasion technique that an attacker could use would be change or scramble the content that the sensor is looking for. This could be accomplished by using some very simple form of encryption. Here is how a simple packet encryption might work:

1st byte of the packet payload is the value to be added to every subsequent byte. If we use 3 then our payload of "0d 0a 5b 52 50 4c 5d 30 30 32 0d 0a" becomes "31 3d 8e 85 83 7f 81 63 65 31 3e" which does not match any of the known signatures. The attacker has now evaded our intrusion detection system. Another twist of this technique could incorporate public key/private key encryption. The private key for the server and the public key for the client could be sent or bundled with the original install. This would render all communication between the 2 hosts unintelligible and undetectable by intrusion detection systems.

Although signature based IDS do provide a useful service to let an administrator know that he/she has been or is being attacked they should not be relied upon. It is far too easy to fool or shut down an IDS machine for them to be utilized as the primary line of defense against intruders. Some recommendations that have been given by Lawrence R. Halme and R. Kenneth Bauer in their article "AIN T Misbehaving: A Taxonomy of Anti-Intrusion Techniques" are to use the following practices in conjunction with intrusion detection:

- Prevention
- Preemption
- Deterrence
- Deflection
- Countermeasures
Intrusion detection should be part of a defense in depth strategy and no single tool or technology should be relied upon exclusively.

There is no comprehensive tool that implements all the above strategies and provides full automation without human interference. Boar is the basis of such a system.

Rule based Systems

Rule based systems are comprised of a database of associated rules. Rules are conditional program statements with consequent actions that are performed if the specified conditions are satisfied. Rule-based systems differ from standard procedural or object-oriented programs in that there is no clear order in which code executes. Instead, the knowledge of the expert is captured in a set of rules, each of which encodes a small piece of the expert’s knowledge.

Rule Based Systems components

To create a Rule-based system, we have to create,

1. A set of facts to represent the initial working memory. This should be anything relevant to the beginning or initial state of the system.
2. A set of rules. This should encompass any and all actions that should be taken within the scope of a problem, but nothing irrelevant. The number of rules in the system can affect its performance.
3. A condition that determines that a solution has been found or that none exists.

Rule-based systems typically consist of a set of rules, a working memory that stores temporary data and an inference engine. The rules encode the domain knowledge as simple condition-action pairs. Each rule has a left hand side and a right hand side. The left hand side contains information about certain facts and objects which must be in order for the rule to potentially fire (that is, execute).

A typical rule for the rule-base might look like this:
IF module_size > 100 AND interface_size > 5 THEN
  error_prone_module
IF error_prone_module AND developer_experience < 2 THEN
  redesign_required

The working memory initially represents the input to the system, but the actions that occur when rules are fired can cause the state of working memory to change. The inference engine must have a conflict resolution strategy to handle cases where more than one rule is eligible to fire.

---

**Theory of a Rule-based system**

The rule-based system itself uses a simple technique: It starts with a rule-base, which contains all of the appropriate knowledge encoded into If-Then rules, and a working memory, which may or may not initially contain any data, assertions or initially known information. The system examines all the rule conditions (IF) and determines a subset, the conflict set, of the rules whose conditions are satisfied based on the working memory. Of this conflict set, one of those rules is triggered (fired). Which one is chosen is based on a conflict resolution strategy.
When the rule is fired, any actions specified in its THEN clause are carried out. These actions can modify the working memory, the rule-base itself, or do just about anything else the system programmer decides to include. This loop of firing rules and performing actions continues until one of two conditions are met: there are no more rules whose conditions are satisfied or a rule is fired whose action specifies the program should terminate.

Which rule is chosen to fire is a function of the conflict resolution strategy. Which strategy is chosen can be determined by the problem or it may be a matter of preference. In any case, it is vital as it controls which of the applicable rules are fired and thus how the entire system behaves.

There are several different strategies, but here are a few of the most common:

- **First Applicable**: If the rules are in a specified order, firing the first applicable one allows control over the order in which rules fire. This is the simplest strategy and has a potential for a large problem: that of an infinite loop on the same rule. If the working memory remains the same, as does the rule-base, then the conditions of the first rule have not changed and it will fire again and again. To solve this, it is a common practice to suspend a fired rule and prevent it from re-firing until the data in working memory, that satisfied the rule’s conditions, has changed.
- **Random**: Though it doesn’t provide the predictability or control of the first-applicable strategy, it does have its advantages. For one thing, its unpredictability is an advantage in some circumstances (such as games for example). A random strategy simply chooses a single random rule to fire from the conflict set. Another possibility for a random strategy is a fuzzy rule-based system in which each of the rules has a probability such that some rules are more likely to fire than others.

- **Most Specific**: This strategy is based on the number of conditions of the rules. From the conflict set, the rule with the most conditions is chosen. This is based on the assumption that if it has the most conditions then it has the most relevance to the existing data.

- **Least Recently Used**: Each of the rules is accompanied by a time or step stamp, which marks the last time it was used. This maximizes the number of individual rules that are fired at least once. If all rules are needed for the solution of a given problem, this is a perfect strategy.

- **"Best" rule**: For this to work, each rule is given a ‘weight,’ which specifies how much it should be considered over the alternatives. The rule with the most preferable outcomes is chosen based on this weight.

**Methods of Rule-based systems**

**Forward chaining**

Rule-based systems, as defined above, are adaptable to a variety of problems. In some problems, information is provided with the rules and the AI follows them to see where they lead. An example of this is a medical diagnosis in which the problem is to diagnose the underlying disease based on a set of symptoms (the working memory). A problem of this nature is solved using a forward-chaining, data-driven, system that compares data in the working memory against the conditions (IF parts) of the rules and determines which rules to fire.
Backward chaining

In other problems, a goal is specified and the AI must find a way to achieve that specified goal. For example, if there is an epidemic of a certain disease, this AI could presume a given individual had the disease and attempt to determine if its diagnosis is correct based on available information. A backward-chaining, goal-driven, system accomplishes this. To do this, the system looks for the action in the THEN clause of the rules that matches the specified goal. In other words, it looks for the rules that can produce this goal. If a rule is found and fired, it takes each of that rule’s conditions as goals and continues until either the available data satisfies all of the goals or there are no more rules that match.

Forward chaining Vs Backward chaining

Of the two methods available, forward- or backward-chaining, the one to use is determined by the problem itself. A comparison of conditions to actions in the rule base can help determine which chaining method is preferred. If the ‘average’ rule has more conditions than conclusions, that is the typical hypothesis or goal (the conclusions) can lead to many more questions (the conditions), forward-chaining is favored. If the opposite holds true and the average rule have more conclusions than conditions such that each fact may fan out into a large number of new facts or actions, backward-chaining is ideal. If neither is dominant, the number of facts in the
working memory may help the decision. If all (relevant) facts are already known, and the purpose of the system is to find where that information leads, forward-chaining should be selected. If, on the other hand, few or no facts are known and the goal is to find if one of many possible conclusions is true, use backward-chaining.

**Soar**

Soar means different things to different people. Soar is used by AI researchers to construct integrated intelligent agents and by cognitive scientist for cognitive modeling. It can basically be considered in three different ways:

1. A theory of cognition. As such it provides the principles behind the implemented Soar system.
2. A set of principles and constraints on (cognitive) processing. Thus, it provides a (cognitive) architectural framework, within which you can construct cognitive models. In this view it can be considered as an integrated architecture for knowledge-based problem solving, learning and interacting with external environments.
3. An AI programming language.

**Unified Theory of Cognition**

A unified theory is a theory which attempts to explain the details of all mechanisms of all problems within some domain. All previous results should be reproduced and explained. A unified theory of cognition has as its domain all of the cognitive behavior of humans.

To assert a unified theory of cognition, one must propose mechanisms by which the results of these human cognitive experiments can be reproduced. The codification and simulation of these mechanisms is tantamount to designing architecture for general intelligence. In this sense, if one wishes to build an artificially intelligent agent using the human as model, the architecture proposed for the agent could be considered a unified theory of cognition.

By presenting architecture for general intelligence as a unified theory of cognition, one can bring additional knowledge to bear on the analysis of sufficiency. A working model - the
human brain - is certainly sufficient to display general intelligence. The assumption is that one should push the limits of the architecture to produce a capability before building some domain-specific or problem-specific tool to overcome the difficulty. Additionally, the set of data represented by experiments in human cognition provide a measure against which one can measure performance, from which one can gain inspiration and insight for further architectural revisions.

**Soar as an AI tool**

Soar is a candidate unified theory of cognition embodied in a computational programming architecture, developed by John Laird, Paul Rosenbloom, and Allen Newell starting in 1982 at Carnegie Mellon University. Its development continues through the efforts of scientists worldwide in the areas of artificial intelligence, cognitive science and human-computer interaction.

For cognitive modeling: Soar's strengths are in modeling deliberate cognitive human behavior, at time scales greater than 50 ms. Example tasks that have been explored include human computer interaction tasks, typing, arithmetic, video game playing, natural language understanding, concept acquisition, learning by instruction, and verbal reasoning. Soar has also been used for modeling learning in many of these tasks; however, learning adds significant complexity to the structuring of the task and is not for the casual user. Although many of these tasks involve interaction with external environments and the Soar community is experimenting with models of interaction, Soar does not yet have a standard model for low-level perception or motor control.

For building AI systems: Soar's strengths are in integrating knowledge, planning, reaction, search and learning within a very efficient architecture. Example tasks include production scheduling, diagnosis, natural language understanding, learning by instruction, robotic control, and flying simulated aircraft.
Soar components

Soar stood for State, Operator And Result because all problem solving in Soar is regarded as a search through a problem space in which one applies an operator to a state to get a result. Over time, the community no longer regarded Soar as an acronym: this is why it is no longer written in upper case.

Knowledge

Soar uniformly represents short-term knowledge as a network of active symbols. Long-term knowledge is a set of condition-action rules. The conditions of each rule form a pattern to match against the active symbol network. When a rule's condition matches, the rule executes by performing its actions. These actions may result in adding (or deleting) symbols in the short-term knowledge structure.

Goal-directed Action

To manage complexity, Soar includes a goal hierarchy, allowing successive decomposition of problems into component sub-problems. Soar includes mechanisms to create new goals automatically in response to a system's long-term knowledge and current situation.

Reaction

Unlike conventional programming languages, Soar does not enforce a serial flow of control. Rather, actions occur any time an associated pattern matches (via a condition-action rule). Multiple rules may fire in parallel, and Soar provides preference mechanisms and automatic subgoals to handle conflicts, if they occur. Soar represents perceptual and conceptual knowledge uniformly in short-term memory. Thus, new actions flow from previous actions and from changes in the external environment. This combination allows Soar systems to engage in interrupt-driven behavior in the same manner as they direct action toward explicit goals. The rule system incorporates the latest pattern-matching technology, allowing rapid processing. Thus Soar is extremely well suited for the development of intelligent systems that must generate actions in time comparable to human decision time.
Learning
Soar includes an automatic learning mechanism inspired by the psychological concept of chunking. Soar compiles sequences of actions into new units of knowledge (chunks) that can 'short-circuit' some reasoning steps when the system faces similar situations in the future. New chunks fit uniformly into a system's existing long-term rule set. Thus, a Soar system can incrementally learn new facts about the world, as well as more efficient representations of its initial long-term knowledge.

Capabilities
Since its development, the Soar architecture has been used to develop a variety of research systems and commercial applications. Examples include NL-Soar (a Soar sub-system that allows programs to learn and interpret natural English text), TacAir-Soar (a synthetic combat pilot, which performs military air missions in real-time distributed simulation environments), KB-agent (a system for automating business claims and high volume processing) and Soarbot (a poker playing agent).

Summary of some unique features of Soar:
- **Problem spaces** as a single framework for all tasks and subtasks to be solved
- **Production rules** as the single representation of permanent knowledge
- **Objects** with attributes and values as the single representation of temporary knowledge
- **Automatic subgoaling** as the single mechanism for generating goals and
- **Chunking** as the single mechanism for learning.

Other great features of Soar are,
- Learning and its integration with problem solving
- Interruptibility as a core aspect of behavior
- Large production rule systems
- Parallel reasoning
- A knowledge description and design approach based on problem spaces
The Soar license is in the public domain and is thus free to download and use.

The following set of points gives the Soar’s structure for reasoning:

1. Perceive the world: Use perceptual models and algorithms to create state descriptions
2. Elaborate the state: Use Soar rules to create new state objects and relations
3. Propose operators: Use Soar rules to create operator objects
4. Select an operator (conflict resolution): Soar architecture selects a current operator, based on preferences created by Soar rules
5. Apply current operator: Use Soar rules to create persistent new state objects and relations
6. Generate actions in the world: Use motor models and algorithms to initiate world actions
7. Go to 1

Soar’s long-term knowledge

All of Soar’s long-term knowledge is organized around the functions of operator selection and application. These functions are composed of four distinct types of knowledge:

Knowledge to select an operator

1. Operator Proposal: Knowledge that an operator is appropriate for the current situation.
2. Operator Comparison: Knowledge to compare candidate operators.
3. Operator Selection: Knowledge to select a single operator, based on the comparisons.

Knowledge to apply an operator

4. Operator Application: Knowledge of how a specific operator modifies the state.

In addition, there is a fifth type of knowledge in Soar that is indirectly connected to both operator selection and application:

5. Knowledge of monotonic inferences that can be made about the state (state elaboration).

Soar represents this knowledge as production rules.
Production rules

Production rules are if-then statements. The ‘if’ part of the production is called its conditions and the ‘then’ part is called its actions. When the conditions are met in the current situation as defined by the working memory, the rule will fire. When the conditions are no longer met some rules retract their actions.

Soar’s execution cycle

Soar’s execution cycle consists of five phases:

*Input:* New sensory data comes into working memory.

*Proposal:* Productions fire (and retract) to interpret new data (state elaboration), propose operators for the current situation (operator proposal), and compare proposed operators (operator comparison). All of the actions of these productions are I-supported. All matched productions fire in parallel (and all retractions occur in parallel), and matching and firing continues until there are no more additional complete matches or retractions of productions (quiescence).

*Decision:* A new operator is selected, or an impasse is detected and a new state is created.

*Application:* Productions fire to apply the operator (operator application). The actions of these productions will be O-supported. Because of changes from operator application productions, other productions with I-supported actions may also match or retract. Just as during proposal, productions fire and retract in parallel until quiescence.

*Output:* Output commands are sent to the external environment.

The execution cycles continue until the halt action is issued from the Soar program (as the action of a production) or until Soar is interrupted by the user.

Soar’s preferences

A preference net is a representation that is a state space in which

- Absolute preferences are identified by links that connect states to acceptable, rejected, best and worst nodes
- Relative preferences are identified by better, worse and indifferent links that connect states to each other.
Soar has an automatic preference analyzer that helps in the resolution of conflicts.

**Impasses**

Impasses signal a lack of knowledge, and therefore, an opportunity for learning. An impasse occurs automatically whenever the knowledge elicited by the current context isn’t enough for the decision procedure to resolve the preferences in working memory to a single change in the context. The language of impasses, like the language of preferences, is defined independently of any domain. When an impasse arises, the architecture also automatically begins the creation of a new subgoal context whose goal is to resolve the impasse. In this way, impasses impose a goal/subgoal hierarchy on the contexts in working memory.

**A Simplified version of Soar Algorithm** (from Soar manual)

```
Soar
  while (HALT not true) Cycle;
Cycle
  Input Phase;
  Proposal Phase;
  Decision Phase;
  Application Phase;
  Output Phase;

Proposal Phase
  while (some I-supported productions are waiting to fire or retract)
    Fire Newly Matched Productions;
    Retract Newly Unmatched Productions;

Decision Phase
  for (each state in the stack, starting with the top-level state)
```
until (a new decision is reached)
   Evaluate Operator Preferences; /* for the state being considered */
   if (one operator preferred after preference evaluation)
       Select New Operator;
   else /* could be no operator available or */
       Create New Sub-state; /* unable to decide between more than one */

Application Phase
   while (some productions are waiting to fire or retract)
       Fire Newly Matched Productions;
       Retract Newly Unmatched Productions;

Learning by Chunking

Chunking is the pervasive architectural learning mechanism. Chunking automatically creates new associations in long term memory whenever results are generated from an impasse. The new associations map the relevant pre-impasse working memory elements into working memory changes that prevent that impasse in future, similar situations. Although basically a deductive or compositional mechanism, chunking serves many purposes. It can integrate different types of knowledge that have been spread over multiple problem spaces. It can speed up behavior by compiling many steps through many subspaces into a single step in the pre-impasse problem space. It can be used as the basis of inductive learning, analogical reasoning. Because it is the only architectural mechanism for changing long term memory, it is assumed to be the basis of all types of learning in people.

Soar and Rule-based system

Soar is the most highly developed rule-based model of human problem solving. Soar is closely related to rule-based system since it uses the same concepts as RBS like, using production rules to query and compare the knowledge base to reach the goal or result. In spite of these similarities soar has many distinct features when compared to a typical rule-based system.
Parallel, associative memory
In Soar all relevant knowledge to the current problem is identified and brought to exist parallel in the memory. Any relevant knowledge is activated by mechanisms which lead to further elaboration of agent belief, proposals to perform different tasks, taking action in the world and so on. Typical RBS instead of activating all rules just chooses the matching rules. If there are more than one rule that match the condition, it resolves the conflict between the chosen ones. This conflict resolution is primarily based on the syntactic features of the rules. For example, RBS might choose the rule that is instantiated with the most recent memory elements. Soar does not need any conflict resolution for rule selection since all relevant knowledge exists parallely in its memory.

Automated belief maintenance
Soar employs computationally inexpensive truth maintenance algorithms to update beliefs about the world. Belief maintenance ensures that agents are responsive to their environments. In typical rule based systems, every item that is added to memory via a rule must be maintained via other rules. This means that every change to an agent’s context must be the result of a deliberate commitment. This commitment is one source of the perception that rule-based systems are generally brittle and inflexible, because they over-commit to particular courses of action. Using automated belief maintenance, Soar does not require deliberation to maintain the consistency of its internal beliefs with the outside environment. All non-deliberate beliefs require no knowledge maintenance or additional commitment.

Preference analyzer
Deliberation in Soar is mediated by preferences, which aides in knowledge selection. Preference mechanism helps agent to express which knowledge it prefers for the current situation. Soar establishes a goal to resolve conflict rather than choosing between rules. Unlike typical RBS, Soar’s decisions are determined by the knowledge available at any given point.

Operator: In RBS, individual rules are the operators compared to Soar’s which is an abstract type, implemented as a collection of rules.
**Rules:** In soar, a single pre-condition rule can be matched with any number of action rules (and vice versa). As a contrast in RBS, the pre-condition and action are combined; hence the agent needs rules for every action, pre-condition combination, leading to a potential combinational explosion in rules.

**Automatic subgoaling:** Soar recognizes conflicts (impasses) in selection knowledge and automatically creates a subgoal to resolve the impasse. A subgoal is a new state, and rule knowledge can be applied to both sub-states and the super-states. Compared to Soar, RBS has only one state. Automatic subgoaling gives Soar agents a meta-level reasoning capability, the ability to reason about their own capability.

**Task decomposition:** Automatic subgoaling enables task decomposition, so the Soar agents can focus its knowledge on a particular level and ignore considerations at other levels. The process of decomposition narrows a potentially exponential number of considerations into a much smaller set of choices. Automatic subgoaling leads to a hierarchy of distinct states; thus, agents can use multiple problem spaces simultaneously, and knowledge can be created that is specific to the unique states. This makes knowledge search simpler (leading to faster rule matching). Moreover, the knowledge base is naturally compartmentalized, providing a scalable infrastructure with which to build very large knowledge bases.

Although Soar uses rules as its lowest-level representation language, the processes Soar uses in conjunction with the rules differ significantly from other rule-based systems. Soar also adds architectural support for operators and problems spaces; higher level, abstract representations not directly supported in other rule-based systems. The result is that Soar systems scale much better (in both performance and the manageability of the knowledge base) and require fewer rules for sufficiently complex applications relative to typical rule-based systems. Given the differences, together with the common preconceptions associated with rule-based systems, it might be appropriate simply to say that Soar is *not* a rule-based system. The alternative argument (historically made by Soar researchers) is that rule-based representations can be used in much more flexible ways than they typically have been.
Tcpdump

Tcpdump is the most widely used open source tool for directly analyzing packets

Overview

Most network administration tools are not based directly on the data being transmitted on a network, but rather on information related to that data. It is sometimes necessary to examine the data packets to diagnose some network problems and also to learn more about network protocols.

The most widely used open source tool for directly analyzing packets is a program called tcpdump, originally written by Van Jacobson. The standard tcpdump, through version 3.4, is maintained and distributed by the Lawrence Berkeley National Laboratory. Tcpdump relies on the pcap library, a system for capturing packets across different operating systems.

What tcpdump can do?

Tcpdump will allow viewing the entire data portion of an Ethernet frame or other link layer protocol and can optionally print the frame header as well (see Figure). In common terms this means tcpdump will allow viewing the entirety of an IP packet, an ARP packet, or any protocol at a higher layer than Ethernet. By default, tcpdump prints packets at the IP layer.

An example of typical tcpdump output looks like this:

```
11:51:46.637811 10.25.71.241.80 > 10.18.0.100.61965: . ack 415 ...
11:51:46.643077 10.25.71.241.80 > 10.18.0.100.61966: . ack 415 ...
11:51:46.644830 10.209.29.151.80 > 10.18.0.100.61961: . ack 458...
11:51:46.653025 10.18.0.100 > 10.7.14.114: icmp: echo request (DF)
11:51:46.658675 10.209.29.137.53 > 10.18.0.100.53454: 46268* - 2...
11:51:46.659970 10.18.0.100.53454 > 10.70.10.79.53: 23134 A? sn...
11:52:24.306670 arp who-has 10.18.1.80 tell 10.18.0.1
```

Each line represents one packet. In **Boar we use this output version**. In another mode, we can ask tcpdump to print all the data within each packet. The output is obviously much longer:

```
16:05:52.209620 10.7.21.77.80 > 10.18.0.100.62532: P 1:236(235)...
   4500 0113 27a4 4000 3f06 d977 0a07 154d
```
This is one entire IP packet.

**Why tcpdump; not any other tool?**

The number of problems that can be solved with the help of tcpdump is limitless. Because it prints detailed information about network traffic, tcpdump is quite an important tool to a network administrator. It gives us a very clear picture of a specific part of the network. For this reason, it is an excellent tool to use, on which a more complicated tool can be built compared to advanced tool like Snort. Snort grabs data like tcpdump does but then analyzes it at a much higher level. It attempts to detect suspicious network traffic of all sorts, including various forms of attacks and probes. Building Boar on top of snort might limit the learning process of Boar which is the ultimate goal.

**Example:**

Imagine a Web browser that is unable to load pages from a particular server; the Web browser
just hangs. Is it a problem with the client, the server, or something in between? If we run tcpdump while loading the Web page, we can watch every stage of the transaction. We can make sure the DNS query for the Web server's hostname is completed; watch the client make the HTTP request to the server, and check to see if the server responds. Regardless of whether the server responds or not, we are now one step closer to understanding the problem.

Tcpdump can also help debug denial of service attacks. If a network is flooded and all other attempts to determine the source or destination of the traffic fail, tcpdump will show the source address, destination address, and type of traffic involved. Even when other methods can pinpoint the traffic, tcpdump is often useful for examining the contents of the traffic should we wish to learn more about the nature of the attack.

There is one catch that can make tcpdump difficult to use: The machine running it must be connected to the network in such a way that it can view the traffic you wish to monitor. This means both that the machine must be connected to the same physical network as the one in question and that the physical network must allow your machine to view the traffic.

**Limitations of tcpdump**

Although tcpdump will display very detailed information about the packets on a network, its view is in some ways limited by the network hardware. For example, a typical Ethernet card will discard packets with an invalid checksum. Therefore, tcpdump will not be a helpful tool for detecting this kind of broken packet on the network. For that, we will need specialized hardware.

Tcpdump is also able to report on only what it finds in the packet. If an IP address is forged in the packet, tcpdump has no ability to report anything else. Tcpdump shows only what the data is and not what it ought to be.

**Understanding dump data**

Even though tcpdump gives us data packets of all protocols, in Boar we just consider the tcp data packets for simplification. Higher versions of Boar can be extended to take into account of all other protocols.
TCP output format

For TCP packets, the output format is:

time source > dest flags sequence [ack ack] win window [urgent] [options]

For example:

...10.7.21.70.80 > 10.18.0.100.34639: P 1461:2921(1460) ack 973 win 63268 (DF)
indicates that 10.7.21.70 on port 80 sent data to 10.18.0.100 on port 34639. The TCP PUSH flag was set, indicated by the "P." The string 1461:2921(1460) gives us information about the TCP sequence number. It indicates that the packet is starting 1461 octets (eight-bit bytes) from the first sequence number tcpdump observed. This is called a relative sequence number. If you would rather view the actual sequence number used in the TCP packet, you can supply the -S argument on the tcpdump command line. The number after the colon is one more than the sequence number of the last byte in the packet, though this number is not really in the TCP header. The number in parentheses, 1460, is the length of the data sent.

The text "ack 973" indicates a TCP ACK was present and that the next expected sequence number in the other direction (data sent from 10.18.0.100 to 10.7.21.70) will be 973. This is also a relative sequence number if the -S flag is not used. Finally, "win 63268" indicates that 10.7.21.70 will accept a TCP window size of 63268 octets. The (DF) represents the presence of the IP don't fragment option. In this example, the urgent TCP flag is not used and there are no extra options to report.

Boar Development Environment

Platform: Linux 9.0
Software: Java (version: )
    Soar (version: )
    Tcl (version: )
Tool: Tcpdump (version: )
In order for Boar to function, java, soar, tcl and tcpdump have to talk to each other.

**Boar Framework**

---

**Software Integration**

**Tcl**

Tool Command Language (Tcl) is an open-source interpreted scripting language invented by Dr. John Ousterhout of the University of California at Berkeley. Scripting languages such as Tcl are interpreted and weakly typed unlike system programming languages such as C++ and Java. This makes them much more flexible (an essential attribute when combining components that weren't originally designed for each other) and provides dramatically faster development and evolution of applications.

Scripting languages typically:

1. Have no separate compilation step;
2. Use dynamic typing for variables instead of static typing;
3. Include high-level data structures like vectors and hash tables as built-in language elements;
4. Provide an interactive interpreter.

Tcl is the integration platform of choice because of its speed of use, breadth of functionality, and enterprise-ready features such as thread-safety, internationalization and cross platform deployment (from 1). In simple terms, Tcl acts as a glue language that can combine several diverse components together.

![Tcl glue together diverse resources](image)

Standard Tcl releases include Tk, an extension package and a cross-platform GUI toolkit.

Tcl provides two interpreter executables:
- `tclsh`: a simple shell containing Tcl interpreter
- `wish`, a simple windowing shell containing Tcl interpreter.

Both `tclsh` and `wish` takes in user commands as input, evaluate them and prints the final results of evaluation.

There are two different approaches to integrate Tcl with a foreign application:
1. First approach is to extend Tcl so that the foreign application’s API functions are available as Tcl commands.
2. Second approach is to embed a Tcl interpreter within the foreign application, so that it can execute Tcl scripts.
Extension Approach
The extension approach adds new commands to Tcl. The new commands are wrappers around the foreign application’s C API functions.

The foreign application’s API functions are usually packaged as a shared dynamic library that’s automatically loaded as part of the Tcl shell environment. Tcl still provides its basic command line and GUI shells, with the new commands available. This approach is a good fit when the goal is to provide scripting capabilities for the functions of an existing API. It’s the approach that Soar uses to expose its C kernel.

Embedding approach
The embedding approach is essentially the opposite of the extension approach.

Instead of adding the foreign application to Tcl, it adds Tcl to the foreign application. The Tcl API includes functions to create a Tcl interpreter, send it commands, and retrieve the results. With this approach, there’s no Tcl command line; the foreign application is in charge. This approach works when the foreign application is a full-blown program rather than a library of API functions.
Soar and Tcl

Soar uses Tcl to implement the Soar user interface. Tcl allows writing extensions to Soar without having to recompile Soar. Tcl also makes it easier to interface Soar with external programs and simulators, and makes it possible for users to write simple simulators to monitor problem solving in Soar.

The command to start Soar,

- Runs wish, Tcl’s GUI interpreter;
- wish loads and executes a Tcl script, start-soar.tcl.

The result is a GUI interpreter for Soar. The Tcl wish executable provides the GUI Tcl interpreter, and start-soar.tcl extends it with the Soar command set. start-soar.tcl also adds Soar-specific features to the GUI.

The process is seamless enough that a Soar user may never realize, and need never think about the fact, that Soar is running on top of Tcl/Tk. The GUI provides menus for the common Soar commands as well as a command line that accepts all Soar and native Tcl commands. For example, a Soar user might enter the command

```
source foo.soar
```

to load the foo agent into the Soar environment, where foo.soar is a text file that contains a Soar rule set. But `source` is actually a native Tcl command.

After loading the file, the Soar user might enter the

```
run
```

command to execute the foo agent. This is a Soar command that Soar added to Tcl, not a native Tcl command. The wish executable without start-soar.tcl loaded returns an “invalid command name” error if you ask it to evaluate `run`. But, from the Soar user’s point of view, the difference between the two commands is invisible.
The Input/Output Links and Tcl

Soar provides hooks for custom Tcl input and output procedures; Soar automatically runs these procedures during its input and output phases. The programmer integrating Soar with an external application can use these procedures to connect to the external application, and to move data into and out of working memory. Boar’s input procedure is iProc(); the output procedure is oProc(). Both procedures are in the boarIo.tcl file.

Soar includes some commands that help with the working memory updates, e.g.

- add-wme - Add a working memory element;
- remove-wme - Remove a working memory element.

Each working memory element has a unique timetag; add-wme returns the timetag of the element it adds. The input/output procedures can save timetags in Tcl global variables, so that working memory elements are accessible for later update or deletion. The syntactic specifics and the documentation (Congdon and Schwamb, and the soar-io.tcl file in the Soar demos directory) is sketchy. Here’s a sample:

```tcl
# Add the clock wme to the input link. Save the timetag of
# the clock wme so we can update during normal input cycles.
scan [add-wme $g_inputLinkId ^clock $g_clock] "%d"
  g_timeTags(clock) # (boarIo.tcl)
```

This code puts a ^clock attribute under the ^input-link attribute in working memory. The new ^clock attribute gets the value of the g_clock Tcl global variable, and its timetag is saved in g_timeTags(clock), a Tcl global hash table.

Tcl global variables are also a convenient way for the input/output procedures and the external application to exchange data. Hash tables are helpful to keep the number of global variables under control.
Java, Tcl and Soar

Soar runs on top of Tcl. So, in order to use Soar with the Java program, Java has to talk to Tcl. As discussed, Tcl offers two approaches to integration: extension and embedding. Because the Java program is a complete, ready-to-run program, the first thing to do is to embed Tcl within it.

Feather

Feather (by Alden Dima) is a public domain Java package that allows a Java application to embed native Tcl interpreters within the same process as the Java virtual machine. This means that a Java program can both call Tcl scripts stored in external files and dynamically create Tcl scripts as Java strings. The result of a Tcl script is returned as a Java String.

Feather is a thin Java Native Interface (JNI) wrapper around the Tcl C API functions for creating and using a Tcl interpreter. Feather makes a Tcl interpreter available as an object in a Java program. `TclInterpreter` class provides two methods: `eval` and `evalFile`. The `eval` method takes a Java string, evaluates it as a Tcl script and returns the result as a Java string. `EvalFile"sources"` takes an external file as a Tcl script and also returns the result as a Java string. Each TclInterpreter maintains its state between method calls.

Once the Tcl interpreter object is instantiated, its methods can be invoked to evaluate a Tcl command, or file of commands, and retrieve the results as a Java String.

Example: (from feather homepage)
Creating an Object:
```java
TclInterpreter interp = new TclInterpreter();
```

Then use it to evaluate Tcl scripts:
```java
try {
    // sourcing a tcl script
    result = interp.evalFile(new File("test.tcl"));

    // calling an individual tcl command
```
result = interp.eval("set a 1");

}  
catch (TclEvalException e) {...}

The build available for feather is basically for a UNIX platform, Robert I. Follek wrote a Linux build for Feather. Boar uses the Linux build version of Robert.

SoarSession
With Feather in place, Java and Tcl can now talk to each other. The final piece is Soar. To get Soar to run via Java and Feather, Soar’s Tcl initialization scripts should be tuned. The wrapper class SoarSession.java instantiates a Feather object and handles the Soar-specific initialization details. To simplify distribution and reuse, SoarSession stores the necessary Soar initialization scripts as string constants. It feeds them to Tcl via Feather.

SoarSession also adds a layer of Soar-friendly methods. For example, to start Soar, load a simple agent, and run it, just create a soar session object and use the member functions:

Code to create a SoarSession object by calling the constructor with appropriate arguments and to load, run and stop an agent.

SoarSession ss = new SoarSession(tclDir, soarDir);
ss.loadAgent(agentFile);
ss.run();
ss.stop();

Java and Tcpdump
As discussed (above) Java (x.java) controls the operation of tcpdump using the bash script file “starttcp” that resides in directory /etc/init.d
The main purpose of starttcp is to start and stop tcpdump, and fine tuning of tcpdump to just capture the required dump data. The tcpdump utility dumps the data into a log file that is read by the java program (x.java).

Start tcpdump:

```bash
$TCPDUMP_PATH/tcpdump -n tcp | sed -e "/ipx/d" -e "/802.1d/d" > $LOG_PATH/log.dat

if [ "`/sbin/pidof $TCPDUMP_PATH/tcpdump`" ]; then
    echo "TCPdump up and running!"
fi
```

Stop tcpdump:

```bash
if [ "`/sbin/pidof $TCPDUMP_PATH/tcpdump`" ]; then
    kill -TERM `/sbin/pidof $TCPDUMP_PATH/tcpdump`
fi
```

**Boar**

**Boar feed – VMSOar**

For this prototype version of Boar, its feed comes from a log file that comes from VMSOar. VMSOar is a cognitive network security agent designed for both network configuration and long term security management. It performs automatic vulnerability assessments by exploring a configuration’s weaknesses and also performs network intrusion detection. VMSOar is also built on the Soar cognitive architecture, and benefits from the general cognitive abilities of Soar, including learning from experience, the ability to solve a wide range of complex problems, and use of natural language to interact with humans. The approach used by VMSOar is very different from that taken by other vulnerability assessment systems. VMSOar performs vulnerability assessments by using VMWare to create a virtual copy of the target machine then attacking the simulated machine with a wide assortment of exploits. VMSOar uses this same ability to perform intrusion detection. When trying to understand a sequence of network packets, VMSOar uses VMWare to make a virtual copy of the local portion of the network and then attempts to generate the observed packets on the simulated network by performing various
exploits. This approach is initially slow, but Soar’s learning ability significantly speeds up both vulnerability assessment and intrusion detection with experience (from VMSoar paper). When VMSoar performs its exploits, the Tcpdump tool running as a background daemon logs these exploits. This log forms the feed for Boar’s analysis.

**Port Scanning**

Port scanning is similar to a thief going through his neighborhood and checking every door and window on each house to see which ones are open and which ones are locked. The TCP/IP protocol suite has 0 to 65535 ports available so, there are more than 65000 doors to lock. The first 1024 ports are called well-known ports and are associated with standard services such as HTTP, FTP, SMTP and so on. Some of the ports over 1023 also have commonly associated services but majority of these ports are not associated with any service and are available for a program or application to use to communicate on.

Port scanning software, in its most basic state, simply sends out a request to connect to the target computer on each port sequentially and makes a note of which ports responded or seem open to more in-depth probing.

If an intruder wants to go unnoticed he might do the scanning in strobe mode or stealth mode. In this mode instead of checking for ports sequentially and within a small time frame, Stealth scan slows the scan, like scanning the ports over a much longer period of time. A **SYN scan** will tell the port scanner which ports are listening and which are not; depending on the type of response generated. A **FIN scan** will generate a response from closed ports- but ports that are open and listening will not send a response, so the port scanner will be able to determine which ports are open and which are not.

When the log created by VMSoar was analyzed by Boar, it found that intruder was trying a **SYN port scan**. Boar rules are set up in such a way that it can detect both the basic scan and stealth scan for SYN scanning.
Boar Port Scan Rule:

If (same destination ip, different port numbers and TCP flag ‘RST’ is set)
then (increment RST count)

//This above rule tries to count all the reject connection communication between the same source and destination. The above rule is not limited to time. So, even stealth scan will not go unnoticed.

If (RST count exceeds threshold limit)
then (give an alert “Port Scan from ‘x’ address”)

Invalid TCP Flag combinations:
The TCP protocol supports the use of six flags: SYN (synchronize), ACK (acknowledge), FIN (finish), RST (reset), PSH (push) and URG (urgent). Certain combinations of these flags determine what type of data packet we are talking about.
Since we have 6 flags, we have 2^6 combinations. But not all combinations are valid.

Consider a case where both the TCP flags SYN and FIN set in the same packet. This is clearly invalid since we do not want to open and close a connection at the same time. Since a SYN packet is used to initiate a connection, it should never have the FIN or RST flag set in conjunction. It is always a malicious attempt at getting past the firewall.
Most firewalls are now aware of SYN/FIN packets. Other combinations include SYN/FIN/PSH, SYN/FIN/RST, SYN/FIN/RST/PSH, etc. These are always a sign that your network is under attack.

Other types of well known illegal packets are FIN (without ACK) and "NULL" packet. A FIN packet should always be accompanied by an ACK bit, since the only reason why an ACK/FIN packet is sent is to tear down an existing connection. A "NULL" packet is a packet with no TCP flags set. Both of these packets also indicate malicious activity. No known TCP stack produces packets with any of the above mentioned TCP flags set for normal activities. If we get an invalid packet as described above, it is always a sign that someone is up to no good.
Boar Invalid TCP flag rule:
Not all invalid flag combinations are captured. This rule is just limited to SYN-FIN, SYN-RST, FIN-RST combinations. In future this rule may encompass the other invalid flag combinations.

If (Flag = (SF or SR or FR))
then (report: Invalid TCP flag combinations set)

Activity on Forbidden port:
In an organization there may be certain rules and regulations of which ports should be used and which ports are forbidden and what the legal activities over the ports are and so on. So it is always handy to define the forbidden ports and make sure if there is an activity on these ports alert the sys-admin.
For example say the FTP port 21 can no longer be used except for special cases. We can put this in a rule,

If (port # is 21)
then (alert : suspicious activity at FTP port)

Boar has another rule to identify if the source of this suspicious activity is from the same subnet. But since, it is possible to spoof the ip address this rule has to be tuned to meet the need.

Sample Source code

To detect activity at forbidden port (22)

This is an elaboration rule which sets “suspicious” working memory element to reflect the ssh port.

sp { elaborate*activity*at*vulnerable*port
    ( state <s> ^io.input-link.Destip.dport << 22 >> )
    -->
    ( <s> ^suspicious at-ssh-port )
}

Proposing an operator alert-admin, if the “suspicious” is set

sp { propose*alert*admin
    ( state <s> )
    ( <s> ^suspicious at-ssh-port )}
Once the operator alert-admin is selected, the following rule fires.

\[
\text{sp} \{ \text{apply*alert*admin} \\
( \text{state} <s> ^\text{io.output-link} <o> ^\text{operator} <o> ) \\
( <o> ^\text{name alert-admin} ) \\
\} \\
\rightarrow \\
( <s> ^\text{operator} <o> + = ) \\
( <o> ^\text{name alert-admin} )
\]

**Future enhancements**

Since Boar is just a prototype, there is a lot of work to be done.

1. Increase the Boar rule-set, so that it catches most of the attacks and effectively protects the system.
2. Instead of standard log feed, Boar should start logging the packets in real-time. This is feasible now, but one problem that Boar faced was how effectively it can log continuous data so that it does not overload the working memory and performs well.
3. Next, step might be to turn on the chunking mechanism, so that Boar can learn and reduce the number of false positives and false negatives.
4. Once Boar is developed it can be totally integrated with VMSoar. Both Boar and VMSoar in combination will be the future cognitive IDS which does not need any human interaction.
5. The other steps of the in-depth defense strategy, preemption, deterrence, etc. can be incorporated.
Bibliography

Alden Dima. National Institute of Standards and Technology. Feather 0.1 Oct. 1999
20 Dec. 2004
http://www.itl.nist.gov/div897/ctg/java/feather/


csis.pace.edu/~benjamin/courses/cs631v/webfiles/VMSoar.pdf

http://www.nottingham.ac.uk/pub/soar/nottingham/soar-faq.html

http://ai-depot.com/Tutorial/RuleBased.html


http://ai.eecs.umich.edu/soar/tutorial/Soar_Part2.pdf


http://ai.eecs.umich.edu/soar/docs/Gentle.pdf


http://codeblitz.com/poker.html

http://www.sans.org/resources/idfaq/

Soar Home Page. 20 Dec. 2004
http://ai.eecs.umich.edu/soar/

http://ai.eecs.umich.edu/soar/docs/SoarRBSComparison.pdf

http://www.phrack.com/

Tcl Developer Site. 21 Nov. 2002. 20 Dec. 2004
http://www.tcl.tk/

Tcl Developer Xchange. Scripting: Next-Generation Software Development. 20 Dec 2004
http://www.tcl.tk/advocacy/whyscript.html