IMPLEMENTING AGENTS FOR INTRUSION DETECTION

Ioan Alfred Letia †, Dan Alexandru Marian ‡
Technical University of Cluj-Napoca, Romania
† letia@cs.utcluj.ro, ‡ marian.dan.alexandru@gmail.com

ABSTRACT
Many advanced techniques have been developed recently to help fight against intrusion. Significant power in this direction can be gained by better taking advantage of the patterns available in the data passing through the network. We have conceived various software agents, distributed over a network, that are able to collect and filter the data and also consider the firewall rules. Preliminary experiments show a significant gain.

Keywords: intrusion, anomaly, agent, firewall

1 INTRODUCTION

While *Intrusion Detection Systems* (*IDS*) have the sole purpose of *sensing* host and network malicious activity, *Intrusion Prevention Systems* (*IPS*) are also required to *act* with the goal of preventing such activities. This means that besides sharing the same detection infrastructure with IDS, IPS try to reduce the probability of an attack by taking a sequence of actions meant to increase the attack difficulty while degrading the network performance metrics ass lower as possible. The most common *intrusion detection* (*ID*) strategies are *anomaly detection* and *misuse detection*. They can be performed either in a *centralised* or a *decentralised* manner. While *anomaly based* IDS build a model of the system under normal operation and label the observed changes as attacks, *misuse based* IDS detect attack patterns using learnt signatures [1]. Hence, *anomaly based* IDS are able to detect novel attack patterns at the cost of producing a larger number of false positives, while *misuse based* IDS have lower false positive rates, but are unable to detect truly novel attacks [2]. Depending on the source of data, IDS can be classified in network based or host based. While *Network IDS* (*NIDS*) collect network device statistics and sense the streams of network data passing though the network's infrastructure, *Host based IDS* (*HIBDS*) rely mainly on host information such as login history, processor and I/O statistics, to detect an attack or a malicious user [2].

The paper presents a way of combining host and network based IDS, and misuse and anomaly ID techniques to enhance packet-filter firewalls. It discusses how real network data collected by a set of sensors and stored in pools of packets can drive the development, evaluation and modification of attack signatures and firewall rules, process that we call evolution. The architecture that will be presented has both sensing and acting capabilities and comprises a suite of intelligent components responsible for collecting and processing data, and finally, generating firewall rules and enforcing them. Successful network attacks, lack of network connectivity and the degradation of network performance metrics are perceived by the system as quantifiable penalties and are *assumed* to be acceptable, while they are not in the case of IPS.

The Erlang [3] programming language has been chosen for building a distributed system, capable of learning and taking actions on the basis of collected information. Due to the offered location transparency, data collection and processing can be done either in a centralised or a decentralised manner. Erlang is a functional language featuring lightweight concurrent processes and process hibernation, hot code loading and supervision trees. Erlang was initially developed for creating scalable systems able to recover from failure [4].

This paper continues the work done on developing an IDS system as a pattern recognition task, using Erlang [5], with the goal of specifying agents meant for enforcing security policies in Local Area Networks. The approach employs information obtained, both from the individual hosts and from the network domain. The agent based approach has several important benefits over the traditional centralised systems. The absence of a single point of failure, the exploitation of local context and the ability to take local decisions in addition to the offered scalability in terms of data collection and processing, are just some of the advantages that agent based systems have. Despite this, the centralised approach has been the main focus of the security world, while the agent based techniques for the computer security domain have been explored in [6][7][8][9]. Therefore, we envision autonomous agents designed to react to network threats and take actions for eliminating or diminishing their effects.

The agents have multiple competing goals and are distributed into an insecure and loosely connected computer network. To simplify the design, we model user needs and actions by an information agent that we call *User Agent*, which is responsible for: retrieving information from the given host or from the local network, and performing actions that a normal user might do. A single *User Agent* runs on a given host, and no collaboration with other *User Agents* from different hosts is assumed. Furthermore, the attacker's behaviour is modeled by an *Attacker Agent* having the purpose of disrupting, biasing or gathering information from the *User Agent*'s host. The *Security Agent* is responsible for the *User Agent's* safety, and for handling the authentication and authorization process.

The agent design has been specified in **GOAL** (**G**oal **O**riented **A**gent **L**anguage) [10] having temporally extended declarative goals [11]. As described in [11], the symbols □ stands for "always", ◊ for "eventually" and ○ for "in next state" and are used to denote operators from temporal logic. For expressing the agents beliefs and desires the symbol **B** for "belief" and **G** for "goal" are employed. *Conditional actions* are expressed using **ϕ** ⊳ **do(*)**, where **ϕ** is a *mental condition*, and * a basic action; time information is introduced using **before** and **until**; for further details please refer to [10] and [11].

2 MOTIVATION OF THE APPROACH

Firewalls, antivirus applications, intrusion detection / prevention systems and audit tools are the most prominent tools for enforcing security policies, detecting and responding to security attacks. Even though great effort has been put into automating the security infrastructure, there is still a need for security professionals. They are supposed to know the best practices, the context in which they apply, and, more importantly, they must be able to predict the impact that such practices might have. In addition, tools have different vendors and this has an impact on compatibility. It is often a practical requirement to use tools from the same vendor hoping that they would be able to cooperate. Yet, the security domain often requires multi-layer and multi-vendor tools. Furthermore, there is the problem of ensuring that security policies are actually enforced throughout the local domain. While domain controllers help in this aspect, there is still the issue of users deliberately try to violate the policy for a gain in processing speed or the problem of delayed responses in the case of online attacks. Hence, one needs to enforce distributed control in the local domain. A suitable means of accomplishing this is by using agents. They offer the means of exploiting existing knowledge and the underlying infrastructure. They offer means of modeling interaction, are able to reproduce, to a smaller scale, the activity of a user and can be replicated, tuned, tested and run in a virtual world without time limits. By being able to generate network traffic, they can be used to check whether a security policy is actually enforced and moreover what would be its impact on the real system. Furthermore, agents could be used to detect security vulnerabilities or weaknesses in an automated manner by using specially crafted attack sequences.
Furthermore, a pool of attacking agents could be used to actively determine the coverage of firewall rules or a pool of agents modeling user behavior might be used to generate relevant network traffic. Moreover, agents might replace, in the near future, security professionals and act as glue between different tools belonging to different vendors. For example, the abstract concept of "drop packet" can be mapped to two different firewalls yet preserve its meaning. Agents can be equipped with the knowledge of a security professional and be able to adapt the actions to the given context, using their perception. Currently there is a major problem related to the gap between what agents can and what it should do. While agents work with abstract concepts, security tools require a detailed knowledge about their inner workings.

The paper intends to model such agents and show how an agent percepts and actions would map to existing security tools and information sources. For example, in an agent specification *src_ip(packet)* would reduce to a logic proposition (true/false) that can be mapped, behind the scenes to an actual value ('192.168.1.1'). The reason why Erlang we employed is that its concurrency model, based on Actor interaction, offers the possibility of running multiple agents and by integrating the sensors a small step is made towards building a security framework.

3 AGENT SPECIFICATION

Since users are able to prioritize their goals and choose inconsistent or unpredictable sequences of actions that often exceed the capabilities of the frameworks developed so far, a *User Agent* is used to model the behaviour of a user. We enforce agents to employ sequences of actions which with their beliefs and goals, and require them to have a consistent goal base. While a *User Agent* models the actions of a hypothetical user, the *Security Agent* models the maintenance tasks and the response to attacks. Its goals are to defend the *User Agent*, prevent and counteract attacks and assess the security of the communication link. The goals of the agents might compete between themselves, as it is the case of the connection maintenance goal and the security goal: dropping network connectivity might ensure protection against attacks and therefore fulfill the security goal, while the communication ensures the flow of information that is needed by the *User Agent* to fulfill the task. We discuss now about the tasks of the *Security Agent* using some simple examples.

Identity management. The *Security Agent* is required to establish the connection with a *Broker Agent* (foreign). The *Security Agent* believes that it knows the *User Agent*'s credentials: $\Box$ **B** username, $\Box$ **B** password, $\Box$ **B** broker_agent. Whenever there is no connection, exactly one such connection is set up:

$$\Box\, (\mathbf{B}\ username \wedge \mathbf{B}\ password \wedge \mathbf{B}\ broker\_agent) \rightarrow connect(broker\_agent,username,password)$$

Logging. The agent's goal is to ensure individual liability. Whenever the *User Agent* sends a request to a broker, the *Security Agent* stores a copy of it for accounting purposes. $\Box\ request \rightarrow \Diamond\ store(request)$ Whenever the broker replies, a copy of the message is also stored. $\Box\ reply \rightarrow \Diamond\ store(reply)$

Resource management. The *Security Agent* acts as a mediator between *User Agents* and its operation follows strict guidelines. Accessing a shared resource (*access_req*) and altering their state require *User Agents* to be authenticated (*auth*) and authorized (*aut*) to perform such actions.

$$\Box valid(id,password) \rightarrow \mathbf{B}\ authenticated$$

Note that due to the fact that GOAL employs linear temporal logic, *valid(id,password)* must reduce to a logic proposition. The rule says that whenever the id (or username) and the password are correct, the *User Agent* is authenticated (*auth*). Since GOAL agents have their internal state represented through mental states, beliefs and goals, the authentication process implies that the agent believes in the given user authentication. An agent is authorized to use a resource (*res*) if it is authenticated and has the permission to use it:

$$\Box(\mathbf{B}\ auth \wedge \mathbf{B}\ has\_permission(resources)) \rightarrow \mathbf{B}\ aut\ (res)$$

An authorized (*aut*) agent must be given access (*ag*) before a given moment can be expressed as:

$$\Box((\mathbf{B}\ \text{aut (res)} \rightarrow \mathbf{G}\ \text{ag(res)}\ \mathbf{before}\ 15{:}00)$$

Note that granting access to resources is viewed as an achievable goal having a temporal hard constraint, 15:00 meaning that access must be ensured up to that time (the starvation problem). Hence, there is a maintenance condition that a *User Agent* eventually releases (*rel*) the resource.

$$\Box ag(res) \wedge \mathbf{B}\ peers\_need(res) \rightarrow \Diamond\ \mathbf{G}\ rel(res)\ \mathbf{before}\ 15{:}00$$

Note that using this specification we are unable to ensure that the other agents would be able to use the resources until the time expires. If the *Security Agent* believes that the *User Agent* has been compromised by an attacker, the *Security Agent* is able to drop the resource usage goal.

$$\Box \mathbf{B}\ compromised \rightarrow \mathbf{drop}(\mathbf{G}\ access\_granted\ (resources))$$

Since the *Security Agent* is required always to run, a portion of the memory should be dedicated for it. This would require killing processes whenever it believes that there is insufficient memory left:

$$\Box\ \mathbf{B}\ \neg\ sufficient\_memory \rightarrow \mathbf{G}\ kill\_process$$

Network connectivity. The *Security Agent* is responsible for ensuring that *User Agents* are entitled to use the computer network. Whenever a *User Agent* has authenticated itself and is authorized to use the computer network, the *Security Agent* performs a sequence of actions to grant the user access (*gua*) to the network.

$$\Box(\mathbf{B}\ auth \wedge \mathbf{B}\ network\_autorized) \rightarrow \mathbf{G}\ gua$$

The actions meant for providing network access are constrained by a conjunction of preconditions expressed as beliefs (ex. **B** *administrator)*:

$$\Box \mathbf{do}(allow\_all\_user\_traffic) \rightarrow \mathbf{B}\ administrator$$
$$\Box \mathbf{do}(allow\_all\_web\_traffic) \rightarrow \mathbf{B}\ regular\_user$$

If no packet is received in a given time frame, the agent should check whether there are some connectivity problems. The agent has the goal of maintaining the connectivity between end-points. The agent believes that it is configured with an IP address **B** *dhcp(ip)*, and that it is able to ping to the outside of the network **B** *ping(outside)*. The strategy is to check if the interface is up, perform a Layer 2 broadcast and see whether there are any replies, perform a Layer 3 broadcast and see whether there are any replies, ping to outside of the local network. In case one of the first three tests fails, the actions performed are to re-enable the interface, and if only the last test fails the strategy is to request a new IP address from the DHCP server.

$$\begin{aligned}
\Pi^{conectivity} = \{&\mathbf{G}\ connected \wedge \mathbf{B}\ layer2\_reply(none) \\
&\triangleright create\_layer2\_broadcast(layer2\_frame), \\
&buffer(layer2\_packet) \triangleright send(layer2\_frame), \\
&(\mathbf{G}\ connected \wedge \mathbf{B}\ layer3\_reply(none) \\
&\triangleright create\_layer3\_broadcast(layer3\_packet), \\
&buffer(layer3\_packet) \triangleright send(layer3\_packet), \\
&\mathbf{B}\ layer2\_reply(local\_network) \wedge \\
&\mathbf{B}\ icmp\_reply(local\_network) \triangleright ping(gateway), \\
&\mathbf{B}\ icmp\_reply(gateway) \triangleright ping(broker), \\
&\mathbf{B}\ icmp\_reply(broker) \triangleright network\_working, \\
&\neg\ \mathbf{B}\ layer2\_reply(local\_network)\ \mathbf{until}\ 30\ \text{sec} \\
&\triangleright reenable(interface), \\
&\neg\ \mathbf{B}\ icmp\_reply(local\_network)\ \mathbf{until}\ 30\ \text{sec} \wedge \\
&\mathbf{B}\ layer2\_reply(local\_network) \triangleright release(ip), \\
&\mathbf{B}\ ip(no\_lease) \triangleright dhcp\_request(ip)\}, \\
\Sigma_o^{connectivity} = \{&\mathbf{B}\ layer2\_reply(none), \mathbf{B}\ icmp\_reply(none)\}, \\
\Gamma_o^{connectivity} = \{&connected\}
\end{aligned}$$

Network protection. The agent has the goal of maintaining the security of the communication link. When the *Security Agent* believes that the system is under attack it has the goal of blocking incoming packets. When this happens it can either simply drop them or explicitly send a reset message, which has the benefit of decreasing the attacker's speed and the price of increasing the workload on the *Security Agent* in case of *Denial of Service attacks* (*DoS*). The same strategy may not apply also for a *Distributed DoS* (*DDoS*) when reset messages have an impact only on a limited number of hosts, and a negative influence on the throughput of the link. Hence, we condition the activation of an action based on a belief.

$$\begin{aligned}
&enabled(silently\_drop(packet)) \\
&enabled(drop\_and\_send\_reset(packet)) \\
&\mathbf{B}\ DoS \triangleright drop\_and\_send\_reset(packet) \\
&\mathbf{B}\ DDoS \triangleright silently\_drop(packet)
\end{aligned}$$

Data security. The agent's goal is that of maintaining the confidentiality of corporate data. If the *Security Agent* believes that the information sent by the *User Agent* infringes a confidentiality constraint, it must drop the packets corresponding to that stream. If the source of a message is not known, the message must be discarded. Furthermore, if the *Security Agent* believes that the alleged source is not the true one, the message is always discarded.

$$\Box\ \mathbf{B}\ \mathit{infringes(packet)} \rightarrow \mathit{drop(packet)}$$
$$\Box\ \mathbf{B}\ \neg \mathit{source(packet)} \rightarrow \mathit{drop(packet)}$$
$$\Box\ \neg\mathbf{B}\ \mathit{source(packet)} \rightarrow \mathit{drop(packet)}$$

Monitoring. The *User Agent* can terminate under three conditions: internal error, attack and normal exit. If the *Security Agent* believes that the *User Agent* has not terminated normally, it has to restart it:

$$\Box\ \mathbf{B}\ \neg \mathit{normal\_exit(user\_agent)} \rightarrow \mathit{restart(user\_agent)}$$

If in three consecutive states the agent believes that the CPU consumption increases (*cpui*), the event is considered to be a sign of an attack.

$$\Box\ (\circ\mathbf{B}\ \mathit{cpui}) \wedge (\circ\circ\mathbf{B}\ \mathit{cpui}) \wedge (\circ\circ\circ\mathbf{B}\ \mathit{cpui}) \rightarrow \mathbf{B}\ \mathit{attack}$$

In the other extreme, high traffic volumes might generate a high number of interrupts that will be perceived as processor idle time. When corroborated with the fact that the traffic is composed mainly from SYN packets, this might be seen as an attack.

$$\Box\ \mathit{processor\_iddle} \wedge \mathit{SYN\_packets} \rightarrow \mathbf{B}\ \mathit{attack}$$

Collaboration with other security tools.

The *Security Agent* can be easily integrated with the existing security tools. One can add external security information in the form of beliefs and map the agent's actions to firewall rules or system call policies. For example, when the system calls performed by an application violate the exiting *systrace* security policy, the host based security tool can add the belief **B** *priviledged_syscall* to the agent's belief state, and the agent will take the decision of allowing or denying the system call based on its mental state, beliefs and goals. While *systrace* can deny any deviation from the established security policy by itself, the technique is not scalable since users might require some functionality so rarely that it does appear in the learning phase. Testing every option that an application may have, in the hope of learning the collection of system calls that the application might make during its lifetime is a challenging task. The agent based approach has the advantage that the required functionality can be split among several components that can take information from multiple sources. The complexity of the overall system is reduced and the individual components can be more easily tailored to the user needs. While considering a mainstream IPS, the anomaly notifications triggered by the system can be integrated in the agent using the aforementioned beliefs. Once the agent has the belief that the system is undergoing a port sweep **B** *reconnaissance_attack*, it uses a maintenance condition to adopt the goal blocking unused ports (*block_up*):

$$\Box\ (\mathbf{B}\ \mathit{reconnaissance\_attack} \rightarrow \mathrm{adopt}(\mathbf{G}\ \mathit{block\_up}))$$

that is fulfilled using the action :

$$\mathbf{do}(\mathit{close\_up}) \rightarrow \mathbf{G}\ \mathit{block\_up}$$

The agent's action, such as *close_up*, can then be translated into firewall rules. As an implementation suggestion, the *netstat* utility can be used to retrieve the currently open ports and the applications that use them. These can be added as agent beliefs such as **B** *port_80_opened*. Using a deny any rule or under the firewall's Closed World Assumption, the open ports remain opened since the agent will assert a firewall rule to let them opened, and the rest of the ports would be closed.

Attacks. Several attack scenarios [12] will be described next and for each one of them a possible agent specification is provided. Just three classes of attacks will be discussed, mainly those: intended to disrupt or degrade the communication flow, those in which the attacker masquerades itself as another agent, and those where the purpose of the attacker is to collect information regarding the victim. These could be expressed as the goals: to disrupt the communication between the other agents **G** *disrupt*, bias ***G*** *bias* or gather information ***G*** *gather_information*. We make the assumption that the attacker believes that the *Security Agent* has software vulnerabilities **B** *software_vulnerabilities*, has unused services running **B** *running(unused_service)* and the corresponding ports are open ***B*** *open(unused_ service_port)*. Furthermore, it assumes that it can take over the control of Secure agent **B** *control(secure _agent).* Knowing that the sub-goals **G** *directed_broadcast*, **G** fragmentation_attack, **G** *ip_options_attack*, **G** *syn_flood*, **G** *reset_attack* entail the goal **G** disrupt the attacking agent selects one of them to accomplish it.

$$\begin{aligned}
\Pi^{\mathbf{directed\_broadcast}} =\{ & \\
& \mathbf{G}\ \mathit{directed\_broadcast} \wedge \mathbf{B}\ \mathit{ip\_dst\_addr(ip)} \\
& \rhd \mathit{change\_ip\_src\_addr(broadcast\_ip)}, \\
& \mathbf{G}\ \mathit{directed\_broadcast} \wedge \mathbf{B}\ \mathit{mac\_dst\_addr(broadcast\_mac)} \\
& \rhd \mathit{change\_mac\_dst\_addr(broadcast\_mac)}, \\
& \mathbf{G}\ \mathit{directed\_broadcast} \wedge \mathbf{B}\ \mathit{mac\_dst\_addr(broadcast\_mac)} \\
& \wedge \mathbf{B}\ \mathit{ip\_src\_addr(broadcast\_ip)} \rhd \mathit{send(packet)}\}
\end{aligned}$$

$$\Sigma_{o}^{\mathbf{directed\_broadcast}} =\{\mathit{ip\_src\_addr(src\_ip)}, \mathit{mac\_src\_addr(src\_mac)}, \mathit{ip\_dst\_addr(dst\_ip)}, \mathit{mac\_dst\_addr(dst\_mac)}\}$$

$$\Gamma_{o}^{\mathbf{directed\_broadcast}} =\{\mathrm{directed\_broadcast}\}$$

$$\begin{aligned}
\Pi^{\mathbf{reset\_attack}} = \{ & \mathbf{G}\ \mathit{reset\_attack} \wedge \mathbf{B}\ \mathit{ip\_src\_addr(src\_ip)} \wedge \\
& \mathbf{B}\ \mathit{ip\_dst\_addr(dst\_ip)} \wedge \mathbf{B}\ \mathit{mac\_src\_addr(src\_mac)} \wedge \\
& \mathbf{B}\ \mathit{mac\_dst\_addr(dst\_mac)} \wedge \mathbf{B}\ \mathit{flags(ack)} \\
& \rhd \mathit{change\_flags\_to(ack,rst)} \\
& \mathbf{G}\ \mathit{reset\_attack} \wedge \mathbf{B}\ \mathit{flags(ack,rst)} \rhd \mathit{send(packet)}\}
\end{aligned}$$

$$\Sigma_{o}^{\mathbf{reset\_attack}} = \{\mathit{ip\_src\_addr(src\_ip)}, \mathit{mac\_src\_addr(src\_mac)}, \mathit{sequence\_number}, \mathit{flags(ack)}, \mathit{ip\_dst\_addr(dst\_ip)}, \mathit{mac\_dst\_addr(dst\_mac)}\}$$

$$\Gamma_{o}^{\mathbf{reset\_attack}} =\{\mathit{reset\_attack}\}$$

Disruption using unknown format or parameter.
The attacker sends packets with some invalid format or parameters the attacker hopping to crash the victim. To prevent this, the *Security Agent* always drops the non-compliant packets that it receives:

$\square$ *has_ip_options(packet)* $\rightarrow$ *drop(packet)*
$\square$*zero_fragment_number(packet)* $\rightarrow$*drop(packet)*
$\square$ *ip_source_routing(packet)* $\rightarrow$ *drop(packet)*

TTL attacks. The attacker sends IP packets with small *TTL* (*Time To Live*) hopping they would expire at the victims interface, to force the *User Agent* dropped them and send an error message requiring computational resources and consuming the agent's bandwidth which leads to a decrease in productivity since fewer resources are available. When receiving such an IP packets, it will believe that either this is a normal event, since it is allowed event, or constitutes an attack *ttl_expire*$\rightarrow$**B**(*attack* $\vee$ *legal_event*). Note that even though, the *Security Agent* could, at a certain point, believe that this was not a legal event $\neg$**B***legal_event*, one cannot infer **B** *attack* using this specification. In contrast, if such packets originate from a single machine then this might be an attack:

same_source_ip $\wedge$ *ttl_expire* $\rightarrow$ **B** *ttl_attack*

since after the first error message was received, the original sender should have stopped sending packets:

$\square$ *ttl_expire* $\rightarrow$ *stop_sending*

Generally though, several attackers are needed for this attack to be successful, which can be specified by dropping the restriction that TTL packets come from the same source IP. The agent will believe it is subject to an attack until there are no expired IP packets: $\square$ **B** *ttl_attack* **until** $\Diamond$ $\neg$ *ttl_expire*

Directed broadcast attack. Packets sent to the broadcast address are expected to be processed by all the networking devices on a local segment:

$\square$*broadcast_address*$\wedge$*dev_on_local_net* $\rightarrow$ *process_packet*

If the source address is also a broadcast address, the networking devices will broadcast their replies and as a consequence the network segment will be filled by high volumes of useless traffic. Packets having the source IP address a broadcast address are unusual at least, hence $\square\neg$ *source_ip_broadcast_address*.

TCP attacks. There is a large range of TCP attacks, among which TCP SYN attacks, TCP SYN-ACK. Firewall cope with this by sending TCP RST (reset) whenever the number of synchronization requests exceeds a given threshold. The advantage of this strategy is that it can prevent resource depletion, but this depends on how fast it can reset incoming connections. Since alternative strategies are available, by incorporating firewall blocking strategies into an agent, we employ a decision process that is more adapted to the context. For example, another strategy is to use TCP keepalives which forces the attacker to respond with the effect of reducing its send rate:

send_keepalive **until** (*received_RST* $\vee$ *received_FIN*)

Expressing the fact that many SYN packets are received is rather awkward, that might be stated:

$\square\Diamond$*synchronization(packet)* $\rightarrow$ ins(**B** *syn_flood*)
B *syn_flood* $\triangleright$ *silently_drop(packet)*
B *syn_flood* $\triangleright$ *drop_and_send_reset(packet)*

The *Security Agent* will then have two actions at his disposal for accomplishing the goal of protecting the system from a SYN flood attack.

Biasing or Collecting Information. The Man in The Middle Attack is meant to bias flow of information between a broker and the *User Agent*. The *Security Agent*'s job is to detect when there are signs of an attacker's presence. For this, suppose that the *Security Agent* is allowed to drop packets only in the case of a Man in The Middle Attack. For an attacker to put itself in between the two agents, it has to guess the sequence number of the TCP connection, must reset the connections between the two endpoints and establish a connection between itself-*Security Agent* and itself-*User Agent*. Suppose that the Attacker accomplishes this and it requests some confidential data from the *User Agent*. The *User Agent* receives such the request and since it believes that the attacker is the broker it sends the secret data.

B *broker_request(attacker)* $\triangleright$ *ins(broker(attacker))*
B *broker(attacker)* $\triangleright$ *send_secrets(packet)*

Since this is an active attack, a rather noisy one, the *Security Agent* can detect it by observing packets with an invalid TCP sequence number. If such a situation occurs, the *drop* action is enabled. Next, if the *Security Agent* believes it has received an IP packet from the attacker who triggered the invalid sequence number condition, the IP packet will be dropped.

$\square$*tcp_sequence_error* $\rightarrow$ *enable(drop(packet))*
B *packet_src(attacker)* $\triangleright$ *drop(packet)*

Port sweep. A common requirement is to always close unused ports: $\square$ $\neg$ *used_port* $\rightarrow$ **do(***close_port***)**.

This can be enforced for well known IP ports (ports smaller than 1024), while for the others, one could specify: $\square\neg$**B** *used_port* $\rightarrow$**do(***close_port***)**.

Furthermore, if a packet addresses an unused port this would indicate a possible vertical scan.

$\square$*unused(port)*$\wedge$ *packet_dest_port(port)*$\rightarrow$**B** *vertical_scan*

If the *Security Agents* were able to collaborate, horizontal scans would be detected in a similar fashion.

4 IMPLEMENTATION

According to the specification, the *Security Agent* uses the belief base as an internal representation of the world and a goal base to express its purpose for which it operates. The agent strives to achieve these goals while exploiting its capabilities and taking into consideration the constraints imposed on it. We view the agent as a layered architecture, in which each layer is an instantiation of the agent for a given role, rather an aggregation of components. Hence, we have an instance of the agent dealing with data acquisition, an agent instance handling goal adoption, belief update and action selection, and one which performs the adopted action in the environment. These are the Sensor, the Reasoner and the *Efector Agent*. The main benefits of this approach appear during the implementation phase. As it will be described in the next sections, the Sensors need to access low level resources to sniff network related data, which usually requires administrative privileges. On the other hand, the *Efector Agent* performs the adopted actions, such as setting a firewall rule, in the environment. It too requires administrative privileges, but by enforcing a distinction between the different layers, a more granular permission control is used. The sensor only has to have permission to access the sniffing interface (or the raw sockets), and the *Efector Agent* has to have elevated rights only for the firewall interface. The deliberative part, performed by the Reasoning agent is then isolated both from the data acquisition and action roles, and has the role of processing the data. The direct consequence is that we have different components on which we can impose different security constraints, rather than on a unique one. Furthermore, this allows us to replicate the *Efector Agent* on all the local domain's computers and let it execute in parallel the actions specified by Reasoning Agent. In turn, we may have a unique Reasoning agent which fuses the data received from multiple sensors and specifies the actions that are to be performed by a single *Efector Agent* (gateway / firewall) or multiple ones (multiple Access Layer devices and / or domain computers). From the implementation point of view the architecture resemblances to a classical management system while on a higher level to an agent. Hence, we benefit both from the well defined semantics of the agents and from the already existing tools and APIs (*Application Programming Interface*).

Layout. The system is built upon a collection of sensors and Reasoner/s. On each monitored host there exists at least one sensor that can be an *Operating System* (*OS*) related tool, whose output will be parsed, a C based application or an Erlang node. Attack signatures, stored in the form of real network packets, are used for the dynamic creation and online testing of firewall rules. Based on these signatures and on host and network statistics, the Reasoner, built in Erlang, creates the firewall rules. Collecting and processing data can be done either in a centralized manner, when all sensors in the network send their observations to a single location, or decentralized when monitored network hosts are responsible for processing their own data and the neighbors' data [13]. The centralized approach has the advantage of not affecting the monitored host's statistics, at the cost of raising privacy and security concerns, while the decentralized approach overcomes these privacy issues, but puts a load on the monitored machine, thus affecting its statistics. In the case of the centralized approach, a single host or a set of dedicated hosts that share the workload are used. Since Erlang has first class functions, *both* approaches can be implemented: in the centralized approach we send the data to the dedicated group of Erlang nodes for processing, while in the decentralised host based approach, functions are sent to operate on the host's data. We have used a centralised approach for collecting data, and a distributed system for processing it.

The major components of the generic IDS/IPS architecture are presented next. We distinguish among *Communication Agents*, *Sensor Agents*, *Processing Agents* or *Reasoners*, and *Efector Agents*. Each such agent fulfills a set of related tasks, using OS tools or applications developed using Erlang and C programming languages.

Communication Agents deal with the secure transmission of data between nodes, by establishing a secure connection through a *Virtual Private Network* (VPN) or port mapping with the help of *ssh*. They also deal with the representation of data shared by applications running on different physical nodes (hosts).

Sensors collect host based information and network related statistics using either *syslog* messages or *Simple Network Management Protocol* (*SNMP*) packets, or by using existing OS querying tools. Sniffing network packets is also their responsibility. Erlang provides support for building UDP, TCP, and SNMP sensors, and a framework for communicating with C or Java applications. Since packet inspection is computationally demanding but is of great importance for detecting attacks, sensors are required to filter the data to be sent for processing. Because support for packet sniffing is not present, the C framework was used for implementing this functionality. The sensors are intelligent, in the sense that they can filter data that this sent for processing.

Reasoners or *Processing agents* are the component where the firewall rules are managed and the amount of needed feedback data is established, by creating special purpose filters. They determine the significance of system wide observations and decide whether an attack is ongoing and what actions should be taken. Communication is done through message passing and their state is kept either locally, or in a distributed fashion (*Mnesia* database). *Efector Agents* execute the actions specified by the *Processing agents* on single hosts or on the overall system. Their most common task is that of applying firewall rules.

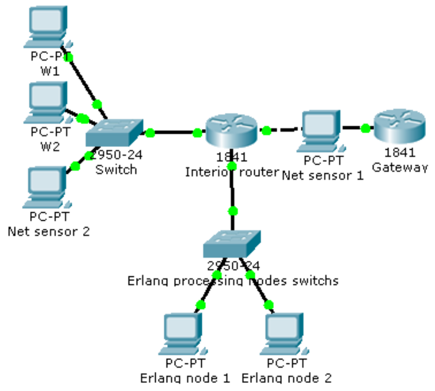

Figure 1: Network Topology

Considering **Figure 1**, between the two screening routers, rightmost handling the communication with the Internet Service Provider, and the leftmost with internal traffic, such as inter-VLAN (*Virtual Local Area Network*) routing, a host is placed in inline configuration. A network sensor, responsible for collecting network related data, an *Efector Agent*, enforcing firewall rules, and a *Communication Agent*, handling the traffic from the sensor to the reasoner, run on it. On the work group switch, on the left side, another such host is connected to the mirroring / span port. All the data collected by sensors is sent to a Processing Agent, composed of a group of Erlang nodes that share the workload. They are connected through a switch, as shown in the bottom of the figure.

Selection and Evaluation of Firewall Rules. The IDS relies on *sensors* to collect data from the local network [13]. These packets are temporarily stored by the Reasoner in two pools of packets: *normal* traffic and *attack* packets, which are used to create the firewall rules. In the learning phase, the packets arriving on the local network interface are put in the *normal packets pool*. During a penetration testing procedure, the resulting packets are stored in the *attack pattern pool*. The normal and attack packets are later combined using a similar procedure with that of genetic algorithms. A new packet is created by combining packets extracted, with a given probability, from each of the pools. Although the combination of packets at arbitrary points should have been allowed, due to efficiency reasons we extract at each layer the header fields and the corresponding payload. We call them features, and let them suffer arbitrary mutation and undergo a crossover process. Layer hashes and packet length fields are not considered features and therefore they are automatically computed for each resulting packet. The TCP sequence and acknowledgement numbers are considered features, but are subjected to the protocol constraints. The genetically evolved packets are then put in the *tentative packets pool* containing packets whose threat characteristics were not established. Firewall rules are generated by an evolutionary process, and the valid ones are set on the local interface. An arbitrary pool is selected, and packet is extracted from it and sent through the specified network interface. Since no firewall rule specifies options such as connection tracking, batch send is used. In the case of connection-oriented protocols, such as TCP, a connection can be initiated by an OS script that either opens a port and listens on it, or makes requests for services. If the packet is dropped due to invalid format, it is not taken into consideration any more. Each rule has six scores assigned: true positive, true negative, false positive, false negative, positive-generality and negative-generality. A false positive is a dropped packet that does not belong to an attack sequence, while a false negative is an attack packet that was not recognized as such. Packets from the tentative pool are used for establishing the generality of a firewall rule, while normal and attack packets are used for the other four. If a rule drops or allows passing a high number of packets from the tentative pool, the rule is considered very general. Dropping a tentative packet increases the negative-generality, while allowing one to pass increases the positive-generality. If a rule affects a limited number of tentative packets, the rule is considered to be very specific. Since firewalls evaluate the rules in a top-down fashion, the generality score helps in the positioning of rules. For example, depending on the default policy we may put rules with high negative/positive generality closer to the top or to the end of the firewall list. Since both *iptables* and, for example *ipfilter*, require setting a default policy, such as drop or pass, the positioning of rules allows packets to be dropped, or passed, more quickly. Furthermore, we may set a threshold for the generality scores, above which the rule is discarded. The *fitness function* for firewall rules is given by a weighted sum of the six scores. The weights are set *a priori*. The *fitness function* for tentative packets is more complex since if a packet matches a firewall rule that has a high generality, it receives a penalty, and if it matches a specific rule then it receives a reward. If the packet does not match any rule and the action is based on the default policy, the score is given by the similarity with the other packets in the tentative pool. If a packet novelty is high, the score it receives is also high. The packets in the normal packet pool can be replaced by new packets sniffed from the local network, if they participated in at least one *crossover* operation or their *novelty*, with respect to other packets, is low. This would ensure that some trails of normal packets would be kept in the tentative packets pool. Since we presume that attack packets are more difficult to obtain than normal packets, *all are kept across generations*. By employing this mechanism a direct link is made between the agent's perception and possible actions that emphasizes the role that of perception plays in the adaptability of the agent.

Collect, Filter & Process Network Data. Gathering network data is a challenging task for software based ID applications since the amount of data that can be captured under heavy load and the latency that this operation induces depend heavily on the OS's network stack implementation [14]. Furthermore, processing and sending network data raises a suite of problems, since large volumes of network traffic are accompanied by a large number of interrupts, serviced by kernel code leaving little processor time available for user space applications. Firewall applications cope with this issue by using a given kernel interface and taking *simple decisions* based mainly on packet *header information*. The criteria used for discriminating packets is given by firewall rules in the form of fixed values, so that as little processing as possible is done. The situation gets complicated in the case of *misuse* or *anomaly* based IDS, since the criteria used for discrimination is either not readily available, or requires expensive computations. In the case of *anomaly* based IDS, OS statistics, such as processor time, memory consumption, number of packets received on an interface or the number of interrupts can help in deciding the sampling time interval. Such information can be collected by parsing the output of available user space OS tools, or by using a kernel module which would retrieve such data from within the kernel. In the case of *misuse* IDS, the *similarity* computation process, usually done in an user space application, requires copying packets from kernel space to user space. If the traffic volume is high, the user space application might not respond in due time and even lose packets. The goal of *minimizing* this traffic can be achieved either by doing *more processing* work inside the kernel or by passing just a *portion* of the packet to the user-space IDS application. By implementing the IDS's basic classifier as a kernel module, we would solve not only the problem of the data transfer, but also the user space IDS scheduling issue. This is done at the cost of *bloating the kernel*. Several issues that make such an implementation rather difficult were identified. Floating point operations are not usually allowed inside kernel space, which would rule out classical Artificial Neural Networks that heavily use them. A solution would be to emulate such operations using integer arithmetic, but the increase in the number of arithmetic operations and in the amount of required memory might cost more than the processing done in user space. Resources available in the kernel are rather limited and the kernel code is expected to accomplish well balanced tasks as fast as possible. If the number of hidden layers and neurons used are kept at manageable sizes, an implementation of an Artificial Neural Network would be possible, depending on the employed algorithm. The same remarks apply for other similar strategies such as Self Organizing Maps and Support Vector Machines. The use of Genetic Algorithms raises memory and performance concerns. A small set of individuals and a careful choice of the evaluation / fitness function are required. The metric used must not require floating point operations, so Manhattan or Hamming distances are possible candidates. Another important issue is related to network packet's *length*. Since they are different, a common denominator must be enforced. If the payload is not used, but just the header, then using handwritten firewall rules and existing dump facilities is probably a better choice. Hence, Bloom filters [15] were used to address such issues, mainly the kernel memory limit and the problem of variable packet payloads. They employ just a bit-string and a set of hashing functions to detect whether an example has been previously encountered (similar to the strategy used by the *ipset* package in case of header fields). The problem with this resides on finding the set of hashing functions which best expresses the novelty or the similarity of packet payloads, since Bloom filters can produce false positives when the bit-string is too small or the hashing functions are not efficient. All of these classifiers require training, testing and an evaluation phase. While evaluation is done using the kernel sensor, the learning and testing phases are accomplished in Erlang since it allows the development of distributed applications and provides the bit syntax that facilitates parsing binary data. Moreover, development and experimentation is done more easily in Erlang. The downside is that multiple machines are needed to *share* the *computational workload*. The advantages of the presented kernel based IDS is the access to low level information and the high processing speed. The disadvantages are the high level of expertise required, not for building it, but for ensuring that it does not become the component causing machines to crash. Before going further a clarifying example is provided, in which packet payloads are used for classification. Suppose decentralised approach for collecting network related data is used in which each host has a Reasoner (the Processing agent) and a *sensor* (classifier running in kernel space). The Reasoner stores the aforementioned pools of packets, normal, attack and tentative, in Erlang *dets* tables. While the sensor implements a simple Bloom filter, the Processing Agent can use more complex algorithms. The Reasoner and the sensor share a set of hashing functions, and the sensor has an entry in the */proc* file systems, allowing both read and write operations. The Processing Agent selects a set of packets from the attack pool, extracts the payload using the bit-syntax and computes the bit-string corresponding to the Bloom Filter which gets written to the sensor's corresponding */proc* entry. When a network packet arrives and a payload is presented, the sensor computes the corresponding hashes and checks the bit-string to see if it the packet is a match. If it is, it stores the whole packet, a part of it or a value into another buffer (but it does not take dropping decisions). The Processing Agent periodically pools the */proc* entry trying to read from it, thus fetching the corresponding data. On the basis of this data, the Processing Node can extract, for example, the corresponding IP addresses and build a firewall rule which will be set by the *Efector Agent*. Similar packets generate similar rules, and therefore the system is responsible for selecting the firewall rules with the highest generality or specificity. Furthermore, the Processing Agent can generate not a single one, but a set of firewall rules, and using the *Efector Agent* set them on the network interface. By sending the same, or similar, packets and checking which firewall rule matched, the efficiency of the firewall rules could be computed. This, can be viewed as feedback and can be used to test the perception and the acting capabilities.

This architecture can also be used for *raw filtering of training datasets*. Suppose that we have a large dataset containing attack patterns. If we are limited to a single host, processing the dataset in Erlang's Processing Agent would take a considerable amount of time. A possible strategy is to send those packets through the local interface, allow the sensor to *alter* its internal bit-string as packets arrive and put the latest version of the bit-string into a buffer for the Processing Agent to read it. If this strategy is also applied on a different interface for a dataset containing normal traffic, the Processing Agent could then fetch both bit-strings, set them on the ingress and egress sensor and pass again through both datasets and store only the interesting packets, those for which the Bloom filters give contradictory answers, thus effectively filtering the datasets.

A strategy that uses the in place infrastructure, namely *Berkeley Packet Filter* (BPF) [16], has also been studied. Mainly, the *BPF* network interface provides access to a virtual machine capable of filtering network packets. Due to its high level of configurability, it allows the development of a 'program' through an *evolutionary approach* that would describe the criteria or the algorithm used for *selecting* packets while running inside the kernel.

Furthermore, BPF allows specifying the portion of the packets to be transferred from the kernel space to user space. The suitability of 'zero buffer copy' [17] for developing IDS is yet another interesting feature that needs to be explored further. The BPF offers a scratch memory and a small instruction set, comprising load operations, arithmetic and logic statements and forward jumps, making it suitable for genetic programming. Using the distributed programming facility offered by Erlang, such BPF 'programs' can be executed at the same time on multiple hosts and the selected packets can be sent, even remotely, for processing. Therefore population based genetic algorithm can be designed in Erlang, tested on real data and the corresponding filters created, since the operations executed by the the BPF's VM can be emulated in Erlang using the provided bit syntax.

Efector nodes were implemented mainly as Erlang C Nodes. They run with superuser rights and their primary task is setting firewall rules and modifying OS configuration files.

User-space *sensors* use a mixture of Erlang and C programming languages. Although they implement similar algorithms, they differ from kernel sensors by the fact that they more flexible. Since, Erlang does not provide libraries for sniffing or sending network packets, we have built an Erlang C Node and an Erlang Port Driver for sniffing or sending network packets. To manage the security risk involved, superuser rights are targeted to the specific resource that need to be accessed. In the case of network packet sniffing, the Erlang sensor can either listen on the network interface itself or parse the output provided by a dedicated OS tool. If a centralized approach is used, on each monitored host there would be an Erlang C Node acting as a sensor. No Erlang VM installation would then be required. But, in the case of a decentralized approach, since Erlang must be present to perform data processing, we would use Erlang Port drivers and require the installation of Erlang. Commonly used when low level access is required or for performing tasks that would take too much time when done in a functional language, *Erlang Port Drivers* provide high performance at a higher risk, because data marshalling takes less time since they run in the *Erlangs's Virtual Machine* (*VM*) address space. But, even though the throughput is much higher, if the application experiences some errors, the VM might crash and the design of port drivers is a complex task. Furthermore, synchronization might pose problems, and corroborated with the fact the library's thread of execution is mapped on VM threads, issues occur. Furthermore, as the documentation states, no blocking operations should be performed. The Erlang version R13B03 tries alleviating these issues and proposes the *Network Function Interface* (*NIF*) which makes development easier. C nodes are the Erlang's solution to creating, porting or integrating C applications. A similar interface exists for the Java programming language. We focused mainly on this approach, since these nodes are standalone applications running outside the Erlang VM. As such, they `should` not crash or affect the Erlang VM, but at the same time they do not benefit from the failure robustness provided by the Erlang system. C / Java Nodes communicate with Erlang through message passing using TCP/IP. The Erlang VM periodically sends heartbeats to see whether a connected node is down. Data marshalling between Erlang nodes and Erlang C / Java nodes incurs a great performance penalty. This is an important aspect in the following scenario: in client-server architecture, the Erlang system sends data for processing to a remote C Node. If the client Erlang VM sends too many messages to the server C Node and data marshalling takes too much time, or in the case the data processing takes too long and the server is not threaded, the C Node will not process and respond to the heartbeat in due time. Thus, the C Node will be considered to be down. A possible solution is to develop the C application outside the Erlang platform and communicate using TCP / IP. In both cases there should be a balance between message size and message number. While taking into consideration that TCP is used, sending a large message seems to be better, with respect to marshalling, than sending many small messages. The Erlang nodes, as well as C or Java Nodes, enter in a thrust domain only if they have the same shared secret, called *cookie*. Erlang nodes are able to execute Remote Procedure Calls (RPC), spawn processes on the local and on remote nodes, run applications on the current host OS or on remote hosts. Communication is done using TCP and it is not encrypted by default. Thus the Erlang VM poses a security risk: if one of the Erlang Nodes, member in a thrust group, is compromised then it can affect all the other nodes. To diminish the changes of affecting the monitored hosts, a *Communication Agent* in the form of a TCP server, to handle the communication task, has been built. This denies many of the Erlang benefits, and was solely used when collecting network and host related data. Taking these security concerns into considerations, we believe that they do not affect our choice to develop an IDS using Erlang, since its primary scope is data collection and processing.

5 Experimentation

Several sensors have been developed for selectively sniffing packets, running either in user space as standalone agents or integrated into the kernel, employing similar selection algorithms, mainly neural networks, genetic algorithms, and Bloom filters. While general purpose Erlang C Nodes, employing either the PCAP or the BPF interface, were implemented as user-space sensors, a genetic algorithm sensor and a Bloom filter were integrated in the kernel (having a */proc* entry on Linux kernel and as new functions on *BSD). When payload features were not considered, the OS's firewall was used for filtering the packets, based on header information. The fallowing example employs *ulog* (similar to *BSD's *pflog*) to collect all incoming HTTP packets:

```
iptables -A INPUT-p tcp--sport80-jULOG--ulog-nlgroup1
```

On FreeBSD extra functions were added to the BPF virtual machine, allowing packets to be processed using these algorithms. For example, the next listing shows a sequence of BPF instructions to select IP packets having a specified source IP address:

```
static struct bpf_insn src_filter[] ={
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x0800, 0, 3),
BPF_STMT(BPF_LD+BPF_W+BPF_ABS,26),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K,mysrcip,0,1),
BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
BPF_STMT(BPF_RET+BPF_K, 0)};
struct bpf_program prg={6,(struct bpf_insn*)&src_filter};
```

Besides the load (*BPF_LD*), jump (*BPF_JMP, BPF_JEQ* jump equal) and return (*BPF_RET*) instructions, *BPF_NEURAL*, *BPF_GENETIC* and *BPF_BLOOM* instructions were added to process the whole packet or starting from a specified absolute position (*BPF_ABS*) expressed in bytes. Since filtering rules are run in kernel space, a user space application that used the BPF interface was then able to select packets to be passed to user space by simply constructing a filter involving calls to these functions. Parameters were passed using *sysctl*. For both the Linux or BSD kernel sensors, kernel memory had to be assigned and it had to be persistent across multiple calls of the processing function. Besides the issues previously discussed in the article, the possibility of a DoS attack to be successful increases with the complexity of the processing algorithms (number of individuals, number of neurons or hidden layer). Another downside of the kernel sensors is that an extra user space sensor is needed for sending the data to the processing nodes. The advantage of such sensors is that they can greatly reduce the amount of data that needs to be sent and processed by the Processing Agents.

For testing purposes the Ubuntu 9.10 and FreeBSD 8.0 were used. The FreeBSD used a custom made kernel and had the *pf*, *pflog* and *pfsync* devices enabled and the *BPF_JITTER* option set, among others. For generating attack traffic *arpon, dsniff, guessnet, hping, hunt, maptha, medusa, ncrack, netdiscover, nikto, nmap, p0f, packit, ping, ptunnel, ssahi, kipfish, xprobe,* and the developed PCAP Erlang Port Driver (for replaying existing attack signatures) were used. During the process of evolving network packets the following were observed: packets with SYN=1, ACK=0 with payload; UDP packets with TCP's fields, for example Sequence Number or Acknowledgement Number; ICMP packets with non-radom / non-fixed large payloads. We assumed that hashes and packets lengths were generated automatically. The genetic algorithm based sensor had a 63% detection rate, the Bloom filter 78%, while the Processing's Agent Artificial Neural Network 67%. These results can be explained by the fact that the machine used for attack was used later for sending legitimate traffic. We consider that this is the correct approach, since in the context of IPv4's NAT the public address of an attacker and of a legitimate user should be the same, and the classifiers role is to distinguish between them. Furthermore, we tried to minimize the needed memory and reduce the incurred delay when processing packets. The Bloom filter's detection rate depends heavily on the dataset and on the bit-string length. Since the payload was primarily used, some legitimate packets as well as attack packets had no payload. These kinds of packets were not taken into consideration. Furthermore, attack and normal packets having the same payload were also not taken into consideration.

6 Related and Future work

There is a vast literature regarding Intrusion Detection Systems (IDS) and an endless pool of algorithms ranging from Artificial Neural Networks and Expert Systems to Artificial Immune Systems [18][19], but there are far fewer taking the declarative approach or the time into consideration. The benefits of augmenting IDS with prevention capabilities, as an Intrusion Prevention Systems (IPS) is, are often overseen also.

While the purpose of this paper is to present ways in which an agent could contribute in enforcing security constraints, we cannot oversee the advantages that an agent based approach could have over a centralised IDS. While having a local view of the system, and being allowed to take local actions, agents can change their environment by performing actions and adapt their perception, for example by reducing the amount of data to process. Since the paper does not handle the case in which agents collaborate, an obvious downside of the local view is the lack of a global perspective provided by data correlation. While this can be achieved locally, such an agent may have problems with distributed attacks. This obvious downside is compensated by the fact that while the centralised IDS must process information from all the sensors, an agent can take advantage of its local view and filter or selectively sample the data or observations needed to update its internal state. While it is possible for an attacker to target the host on which the IDS runs, in the case when agents are used it would have to attack all of them. An agent based approach playing the role of an IPS can also adapt to the environment on which it runs, to the specific hardware devices for example, with respect to commands or configuration files. Rather than a complex centralised system controlling the whole system, the agent based approach ensures that the control is distributed across the whole network. While the use of agents for IDS [20] or for enforcing policies is already present in the literature, the use of agents with declarative goals playing the role of IPS is to best of our knowledge novel. Although the paper presents only the design of such agents, we believe that their implementation is feasible and the offered flexibility outweighs the complexity of their implementation. Furthermore, the paper shows how the perception of the environment can be achieved and used to adapt the actions. The next step would be to establish the collaboration between such agents, to achieve the advantage that distributed collaborating agents would have over them [21]. While in the case of the algorithms described in [18][19], explicit or implicit thresholds or quantifiable values are present, in the case of the agents described in this paper variables reduces to boolean values at a higher level. Therefore, the actual data processing operations must reduce to boolean values. We believe that an implementation could overcome this issue by the use of internal buffers.

The agents described in this paper use temporally extended goals, in the context of [11]. Similar uses of temporal logic in Intrusion Detection are explored in [22]. A characteristic of IDS based on temporal logic is their use of past experience (past temporal logic). Therefore what distinguishes the presented agent from them is the fact that a GOAL agent with temporally extended goals is allowed to look only into the future and it can maintain past experiences only through its belief base. To the best of our knowledge, creating network packets through an evolutionary approach for testing the suitability of firewall rules has not been addressed. The use of genetic programming or genetic algorithms for evolving rules for IDS has been investigated in [23][24][25][26] with an overview regarding the use of a distributed system in [13]. To the best of our knowledge, evolving firewall rules and testing their suitability has not yet been done for BPF based firewalls. We are unaware of the existence of open source kernel modules designs playing the role of an IDS, although we think that commercial applications are using this approach. The creation of `programs` by an evolutionary approach targeting buffer overflow was investigated in [27]. We consider that the idea of evolving 'programs' using the BPF instruction set by an evolutionary approach for reducing the amount of data transfer between kernel space and user space is novel. Furthermore, the use of the Erlang programming language for collecting and processing of network and host related data doesn't seem to be covered in the existing literature. The use of an evolutionary approach for developing BPF filters remains to be further investigated in a future paper. Such a strategy is promising, but currently we are not satisfied by the assumptions that we had to enforce.

7 Conclusions

IDS rely on the collected data when deciding whether the network is under attack. Obtaining and processing relevant data is changeling since it depends on many factors. The paper describes the design of kernel and user space sensors and presents a ways in which the data can be filtered and processed. Based on the data received from the sensors, the paper describes a way in which the firewall rules can be built by evolutionary means. Equipped with perception and given a set of actions that translate to firewall rules, the paper presents the design of a *Security Agent* that exploits existing security tools, such as the firewall, in its endeavor of protecting the *User Agent*, modeling a common user, from an attacker. The *Security Agent* is put in a realistic environment that illustrates the interaction with the attacker and the *Security Agent* and several scenarios are depicted.

To simplify its implementation, we consider it as having a layered architecture, where each layer represents an instance of the agent fulfilling a specific purpose. For implementing the components of the system, a mixture of the Erlang and C programming languages has been used. A description of how existing OS tools can help and an insight on how an IDS can be enhanced to the level of an IPS was also described. The benefits and tradeoffs of different approaches along with their motivation were also presented.

ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for the very useful comments. Part of this work was supported by the grant ID 170/672 from the National Research Council of the Romanian Ministry for Education and Research.

8 REFERENCES

[1] S. Chebrolu, A. Abraham, and J. P. Thomas: Feature deduction and ensemble design of intrusion detection systems, Computers & Security, vol. 24, pp. 295-307 (2005).

[2] S. X. Wu and W. Banzhaf: The use of computational intelligence in intrusion detection systems: A review, Applied Soft Computing (2010).

[3] J. Armstrong: Programming Erlang: Software for a Concurrent World, The Pragmatic Bookshelf (2007)

[4] J. Armstrong: Making reliable distributed systems in the presence of software, Ph.D. dissertation, The Royal Institute of Technology (2003)

[5] I. A. Letia and D. A. Marian: Embarking on the road of Intrusion Detection, with Erlang, In: Proc. of the 10th International Conference on Development and Application Systems (2010).

[6] M. Crosbie and G. Spafford: Defending a computer system using autonomous agents, In: Proc. of the 18th National Information Systems Security Conference, vol. 2, pp. 549–558 (1995).

[7] N. Basilico, N. Gatti, and F. Amigoni: Developing a deterministic patrolling strategy for security agents, In: WI-IAT ‘09, pp. 565–572 (2009).

[8] F. A. Barika, N. E. Kadhi, and K. Ghedira: Agent IDS based on Misuse Approach, Journal Of Software (2009).

[9] N.Jaisankar, R.Saravanan, and K. D. Swamy: Intelligent intrusion detection system framework using mobile agents, International Journal of Network Security & Its Applications (2009).

[10] F. de Boer, K. Hindriks, W. van der Hoek, and J.-J. Meyer: A verification framework for agent programming with declarative goals, Journal of Applied Logic (2007).

[11] K. V. Hindriks, W. van der Hoek, and M. B. van Riemsdijk: Agent programming with temporally extended goals, In: Proc. AAMAS ’09 (2009).

[12] J. Mirkovic and P. Reiher: A taxonomy of ddos attack and ddos defense mechanisms, SIGCOMM Comput. Commun. Rev., (2004).

[13] A. Abraham, R. Jain, J. Thomas, S. Y. Han: D-SCIDS:Distributed soft computing intrusion detection system, Journal of Network and Comp. Applications, vol. 30, no. 1, pp. 81–98 (2007)

[14] Q. Li and K. Macy, Optimizing the BSD routing system for parallel processing, In: Proc. ACM SIGCOMM, pp. 37–42, (2009).

[15] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood: Deep packet inspection using parallel bloom filters, IEEE Micro, vol. 24, no. 1, pp. 52–61 (2004).

[16] S. McCanne and V. Jacobson: The BSD packet filter: a new architecture for user-level packet capture, In: Proc. USENIX Winter 1993 (1993).

[17] R. N. M. Watson C. S. J. Peron: Zero-copy bpf buffers, FreeBSD Developer Summit (2007).

[18] S. X. Wu W. Banzhaf, The use of computational intelligence in intrusion detection systems: A review, Applied Soft Computing (2010).

[19] V. Chandola, A. Banerjee, V. Kumar: Anomaly detection: A survey, ACM Comput. Surv. (2009).

[20] G. Helmer, J. S. K.Wong, V.Honavar, L. Miller: Lightweight agents for intrusion detection, Journal of Systems and Software(2000)

[21] P. Kannadiga and M. Zulkernine, Didma: A distributed intrusion detection system using mobile agents, In: Proc. SNPD-SAWN (2005).

[22] P. Naldurg, K. Sen, and P. Thati: A temporal logic based framework for intrusion detection, Proc. 4th IFIP WG 6.1 (2004).

[23] W. Lu and I. Traore, Detecting new forms of network intrusion using genetic programming, Computational Intelligence, Vol. 20, (2004)

[24] W. Li: Using Genetic Algorithm for Network Intrusion Detection. Proc. of the United States Department of Energy Cyber Security Group 2004 Training Conference, pp. 24-27 (2004).

[25] A. Abraham and C. Grosan: Evolving intrusion detection systems, Genetic Systems Programming: Theory and Experiences, vol.13, (2006).

[26] H.G.Kayack,A.N.Zincir-heywood, M. Heywood, Evolving successful stack overflow attacks for vulnerability testing, In: Proc. ACSAC, pp.225-234 (2005).