Embarking on the road of Intrusion Detection, with Erlang

Ioan Alfred LETIA[1], Dan Alexandru MARIAN[2]
[1]*Department of Computer Science*
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
letia@cs.utcluj.ro
[2]*Technical University of Cluj-Napoca*
Department of Computer Science Cluj-Napoca, Romania
marian.dan.alexandru@gmail.com

***Abstract* — Intrusion detection techniques are indispensable for the security infrastructure in order to detect threats before damage is produced. New methods have been conceived using advanced mechanisms, some of them biologically inspired, but all need some kind of information fusion. To be able to deploy these methods some functionalities are required for processing the huge amount of data passing through the network environment. We have been developing such functionalities in the Erlang programming language and are presenting them in this paper.**

***Index Terms* — Computer maintenance, Computer network reliability, Computer network security, Functional programming, Network operating systems**

I. INTRODUCTION

Several methods have been conceived for intrusion detection [1], [2], [3] most of them based on advanced computational intelligence. Due to the huge amount of data passing through the computer networks, it is of paramount importance the ability of efficient processing. We are using Erlang, a concurrent programming language, developed for distributed system, to better cope with the primary tasks of data extraction on the path to intrusion detection. To the best of our knowledge, the functionalities of Erlang have not been exploited yet in intrusion detection.

A. INTRUSION DETECTION

One of the most valuable assets of a company is its communication network (according to the "Data Breach Investigations Report", 2009)[1]. Intrusion detection systems (IDSs) are systems built using software applications, hardware devices, or both, that try to detect potential external or even internal threats to the communication network. They can be classified into two different, but complementary classes: signature based and anomaly based.

1) Signature based systems

These systems classify the information extracted from the communication network, under operation, while relying on a signature database. This strategy is used by many commercial applications since it provides a high detection rate for known threat patterns.

2) Anomaly based systems

Anomaly based systems [4] use network statistics and behavioral analysis to specify if the network is under attack, or not. Since more computations need to be performed or more observations must be collected, anomaly based IDS may provide less throughput than signature based IDS. But at the same time, anomaly based IDS are able to detect new threat patterns and adapt the detection system accordingly.

In this paper we present some functionalities of the Erlang programming language, suitable for building an anomaly based IDS. While the throughput is smaller than the one provided by the C or C++ based IDS, we believe that the features Erlang applications provide, such as scalability, fined grained parallelism, failure contention in lightweight processes, and the ability to perform live updates, are more important for intrusion detection.

B. WHAT ERLANG PROVIDES FOR INTRUSION DETECTION.

Erlang is a dynamically typed concurrent functional language [5]. The code unit of an Erlang program is the module, which is a collection of functions. A module has a public interface given by the functions it explicitly exports [6]. In order to use its functionality, a module has to be compiled/interpreted and loaded in the Erlang Virtual Machive (VM)[2] . Erlang mimics inheritance through the use of parameterized modules [7], and module behaviors (similar to abstract methods of an abstract class in Java).

A function is identified by *name* (the function name is an atom, e.g. myfunction or 'The name of my function') and *arity*[3]. Each function has to return a value [6] that can be an atom, a number, a list, a tuple, a globally unique reference, a process identifier, or even a function due to the offered support for higher order functions [8].

The *Erlang Virtual Machine* (**VM**) supports the creation of a large number of *lightweight processes* by *spawning* functions [8], which can later be explicitly put in a *hierarchy* of processes called ***supervision tree***: a

[1]http://www.csoonline.com/documents/pdfs/2010CyberSecurityResults.pdf

[2] In embedded mode it is loaded at start-up, while in interactive mode only when a module's function is referenced.

[3] Two functions with the same name, but different number of arguments are considered to be different. The type of the parameters does not count. To distinguish among these, one has to specify the arity when exporting the function -export([myfun/1,myfun/2]), or when using it F=fun(X)→fun myfun/1 end.

monitoring process can trap exit signals of the processes it monitors and take the appropriate actions.

Although error handling structures are present, the Erlang programming philosophy is to let processes crash and spawn fresh ones [5], [9].

Each process has a globally unique *Process Identifier* (**PID**) across different nodes in the same Erlang system. The PID is used to address a specific process on any *node* in the Erlang system, thus offering *location transparency*. A PID can be globally registered using an *atom* as an *alias*.

Processes ***share no memory*** and *communicate through* ***message passing***. A message is guaranteed to arrive to the destination, unless the case in which the destination is down[4]. Each process has an associated *message mailbox*, and messages are taken out using the *selective receive* construct [8].

In order to reduce memory consumption (allocated dynamically), processes not often used (e.g. processes waiting for client requests) can explicitly request to be put on ***hibernation*** until a new message arrives [10].

The Erlang scheduler supports ***soft real-time deadlines*** and has priority classes [11], [10]. Garbage collection is automatically carried out, but only when this is actually needed [10].

Erlang is a language that was designed for Telecommunication applications [5] and has features that make it useful when building an IDS.

Although Erlang is a ***dynamically typed*** functional language, along with lists and tuples it has the ***binary type*** which helps in representing raw data in a memory efficient way. The built-in bit syntax and the regular expression module allow analyzing network packets in a simple manner.

The failure contention in lightweight concurrent processes along with the supervision tree ensure that crashed Erlang processes will be automatically re-spawned. The ***hot code loading*** mechanism allows updating the detection system without the need of shutting it down or restarting it later.

Distributed applications can be easily built since Erlang's inter-process communication is done using message passing and the enforced location transparency allows distributing the processing and the communication load across different nodes in the same *thrust group*. It does not matter on which node an Erlang process runs, since it is uniquely addressed using its PID or using a globally registered alias.

The *hibernation* mechanism, allowing process to be put in a state in which they consume as less resources as possible, is a feature that IDS could benefit of. Practically, it makes possible to have many instances of the same classifier or many types of collaborative classifiers available at the same time.

What Erlang solves are *robustness* and *scalability* issues. It was our choice since it allowed us to experiment with different algorithms and easily distribute the processing load. Furthermore, we were also encouraged by its ***fault recovery*** promise.

[4] In case of communication between two Erlang nodes, a "reliable" point-to-point connection is established using TCP.

At the same time, Erlang has issues that make it less recommended: limited support for encrypted communication and permission control, and the absence of a built-in function for sniffing network packets.

For addressing issues that appear during application development, the Erlang system offers frameworks for developing C or Java applications.

We have used the C based approach to develop a packet sniffer, in the hope that this would increase the speed and will provide better access to Input/Output functions. The development was difficult at start, due to the scarce documentation.

We approach the IDS design as a pattern recognition task [12]. The paper describes possible ways of gathering observations, and later processing them. In the next section we will present common issues encountered when building an IDS and we provide simple solutions.

II. THE RECOGNITION PROBLEM IN INTRUSION DETECTION

The notion of *event* and *observation* are central to our discussion. We consider an event to be an act that changes the *state* of the system. It can be a software event, a program crash or a security constraint violation, or a hardware event such as device failure.

Sensors capture the changes in state of the system's components in the form of *observations*. They try to capture the occurrence of such events and their impact on the overall system.

Launching a program in execution or receiving a network packet are *events*, while consulting the network interface statistics or the list of running processes are considered *observations*. Based on these observations, the IDS will decide whether the system is under attack or under normal operation.

Obtaining such observations should not change the system's state. This is a desideratum that cannot be fulfilled in practice, since making an observation requires an application to be run: collecting processor statistics requires a program, such as **ps**, to be run for a short time period.

This is a kind of *noise* induced by the observation. Anomaly based IDS are sensitive to this, since they require building a model of the system under normal operation, and later using it for comparisons. As a consequence, Anomaly based IDS have issues regarding false negative and false positive detection rates. Commercial *signature* based IDS have better false negative rates since they just try to identify an already discovered signature.

One way to overcome this is to gather observations from multiple hosts, and hope that the induced noise is the same and its overall significance can be decreased.

C. Data collection

Gathering system wide observations is challenging not only because Operating System's (**OS**) querying tools differ and observations are affected by noise, but also because computers are used for different sets of tasks. Furthermore, different application and maybe even different OS, are used for accomplishing even the same task.

Observations might incur a *security* and a *privacy* risk,

since making observations sometimes requires administrative privileges. It is one thing to run an application with administrative privileges on user's workstation, and a different thing to run it on the network's gateway.

In the hands of an attacker the required observations might show the vulnerable points and give the means of attacking the network or the common users [13]. If these observations are sent to a remote site for processing, this risk is further enhanced.

Valuable sources of information are **syslog** messages. They are *unencrypted* messages that capture, more or less, the significant events of the system and are ranked accordingly. Applications offering this service are available on all major OSs.

Developing a **syslog** sensor is a simple task in Erlang (Listing 1). Listing 2 shows two possible outputs.

The sensor follows a Publish-Subscribe pattern. The UDP sensor is created by spawning the **init** function located in the module **udp_sensor** with the arguments ***Port***, the UDP port on which it will listen, and the a priori list of listening processes, ***Listeners*** (Line 4). New listeners can be added by sending a message to the sensor, in the form of a *tuple* composed from the atom **add_receiver** and the receiver's own PID (Lines 26-27,17-19).

The actual UDP sensor is created using the **gen_udp** interface (Line 7), by opening Port 514 (Line 50), with a buffer of 16384 bytes (Line 7)[5]. The ***Socket*** variable will bind to the corresponding socket identifier, and it will be used in the listen function to selectively receive messages from that socket.

The message will be kept in *binary* representation (Line 7). The sensor waits for a message to arrive in a loop (Lines 12-23), and uses a *selective receive* to fetch either a **syslog** message (Line 14), or a request to add a process as a subscriber (Line 17). After receiving a message the process goes into ***hibernation***.

If a message does not match any of these patterns, the message is kept in the internal *mailbox*. Unmatched messages increase the memory consumption, so to overcome this issue the ***Error*** variable (Line 20) was used. It will match and bind to every message, at a time, thus removing from the queue the messages that were not matched by the previous two patterns.

All the operations that regard the process of getting a message from the network are *transparent* to the user.

A subscriber process receives a tuple of the form {udp,Socket,Host,Port,Binary}, where the atom ***udp*** designates the transport protocol, the variable ***Socket*** the OS socket used, the ***Host*** variable designates the IP address of the sender, the ***Port*** variable the port number on which the packet has been received and finally the actual message encoded in binary form.

Listing 1. UDP

```
1.  -module(udp_sensor).
2.  -compile(export_all).
3.  sensor(Port,Listeners)->
4.        spawn_link(?MODULE,init,[Port,Listeners]).
5.
6.  init(Port,Listeners) ->
7.     case gen_udp:open(Port,[binary,{recbuf, 16384}]) of
8.          {ok,Socket} -> listen(Socket,Listeners);
9.          {error,eacces} -> exit('Raised privileges needed')
10.    end.
11.
12. listen(Socket,Listeners) ->
13.    receive
14.       {udp,Socket,_Host,_Port,_Bin} = Message ->
                    send_message(Message,Listeners),
15.                 erlang:hibernate(?MODULE,listen,
16.                                  [Socket,Listeners]);
17.       {add_receiever,ReceiverPID} ->
18.                 NewList = [ReceiverPID|Listeners],
                    erlang:hibernate(?MODULE,listen,
19.                                  [Socket,NewList]);
20.       Error -> io:format("Error : ~p~n",[Error]),
21.                listen(Socket,Listeners)
22.    end.
23.
24. send_message(Message,Subscribers)->
25.    [Subscriber ! Message || Subscriber<-Subscribers].
26.
27. add_subscriber(Sensor,ReceiverPID)->
28.     Sensor ! {add_receiever,ReceiverPID}.
29.
30. subscriber_example(Name)->
31.    receive
32.         Message -> io:format("Subscriber:
33.                    ~p~n~p~n",[Name,Message]),
34.                    erlang:hibernate(?MODULE,
35.                 subscriber_example,[Name])
36.    end.
37. subcriber_create(Name)->
38.    spawn( fun()->
39. subscriber_example(Name)
40. end).
41. test(Port)->
42.    SubscriberPID = subcriber_create("S1"),
43.    SensorPID = sensor(Port,[SubscriberPID]),
44.    AnotherSubcriberPID = subcriber_create("S2"),
45.    ?MODULE:add_subscriber(SensorPID,
46.                           AnotherSubcriberPID),
47.    SensorPID.
48.
49. test(Port,TestPID)->sensor(Port,[TestPID]).
50. test()-> test(514).
```

Listing 2. Output

```
{udp,#Port<0.503>,
   {192,168,0,1},
   514,
   <<"<110>Fri Feb 12 13:09:32 2010 router System Log: Blocked
incoming TCP connection request from 79.136.116.120:20867 to
188.24.20.60:20097">>}{udp,#Port<0.503>,
   {192,168,0,1},
   514,
   <<"<110>Fri Feb 12 13:09:32 2010 router System Log: Blocked
incoming TCP connection request from 79.136.116.120:20867 to
188.24.20.60:20097">>}
```

[5] Administrative privileges may be needed.

Since two functions with the same name, but different arity are distinct, the application can be tested by providing the Internet port number used for listening (Line, the port number and a subscriber PID, or using no argument.

SNMP sensors are another important source of information. The *Simple Network Management Protocol* (**SNMP**) is used for managing and gathering (querying) information from network devices. SNMP support is commonly available only for business rated network devices and not for cheap consumer ones. The Erlang system supports SNMPv1, SNMPv2c, SNMPv3[6]. One could use this system to get notifications from Erlang Nodes, to see whether the sniffing agent at a remote site is running or to get information from networking devices (routers) or hosts.

Abstract Syntax Notation 1(**ASN1**) and *Interface Description Language* (**IDL**) are also supported. ASN1 is used for defining the Management Information Base (MIB) object of the SNMP, and IDL is a way to communicate with foreign applications by establishing a common interface.

As in any other language, running external applications and fetching their output is possible in Erlang. Parsing is facilitated by the functional nature and the Prolog-like syntax of Erlang. The applications will run with the privileges of the Erlang VM.

A group of Erlang **nodes** that share a *common secret*, known as the **cookie**, form a *thrust group*. Nodes in a thrust group communicate, by default, using *unencrypted* message passing. Combined with the fact that a node can create processes, make Remote Procedure Calls (RPC) and execute foreign application on a different node, in the same thrust group, Erlang is rather *insecure*. Furthermore, functions are *first class objects* and can be sent across nodes, so if the security of an Erlang node is compromised then the entire group is under threat.

The Machine module (Listing 3) shows some *Built-in Functions* (**BIFs**) for changing the *thrust* group secret cookie and provides an insight on how Erlang could be used to collect relevant information from the observed computers.

Three use case scenarios are presented: getting a list of files, getting network interface information and retrieving statistics about host processes.

Since an arbitrary length string of characters delimited by ' ' is interpreted as an Erlang *atom*, we can use a hole sentence as a function name. This improves code clarity, since one could build a program that looks as a text (the '*What are the Erlang Runtime Statistics*'() function). The downside is the increase in code size. This strategy was used when getting network interface information or Erlang VM statistics (*'Parse command output disregarding first line*'("**netstat -i** ").).

In contrast, succinct programs can be written using function composition, but this has a direct impact on code clarity. This strategy was used in case of retrieving information about recent file changes and OS processes.

For running an application we use the function **cmd** from module **os** to run the command ***Cmd***. In the case of function **parse_cmd_out** the output resulting from running the external application is split into lines and then into specific components using regular expressions.

re:split will return a list of matched elements {return,list}. Since the *trim* option was used, if the last element is the empty list then it is removed.

The *Abstract Syntax Tree* (**AST**) was manipulated using the construct **(**Function**)(**Arg**)** to perform the ***function composition*** of the **split** and **os:cmd** functions.

Last but not least, another important issue regards network packet sniffing. Erlang does not provide any function calls for this. Therefore we had created a C Node agent, using the well supported **PCAP** library. Although it would have been easier to use an **Erlang Port Driver**, sniffing packets requires administrative privileges and therefore the Erlang VM should have been run with such privileges. The **C Node** approach was chosen due to security considerations, in order to ensure isolation.

D. Data processing

An IDS requires data to be represented in a convenient format. Here we discuss data pre-processing for an Erlang based IDS.

The list, the tuple, and the binary data types are the most common. Therefore, the large majority of the Erlang's libraries work on lists and tuples, rather than binaries. They can be transformed into lists using **binary_to_list** (Listing 4), but this increases the memory consumption. Strings are represented as lists of integers, so they also consume more memory[7] than in other languages.

In Listing 4 we illustrate the bit **syntax** that Erlang offers. A sample run is provided in Listing 5. The **flooding** function (Line 13) tests whether the *source* address is a *broadcast* address. This situation should not happen in practice. If there is an Ethernet packet matching the first clause, **true** will be returned, otherwise the second clause (Line 17) will match returning false.

In the second test (Line 18) we check whether the *source* and the *destination* addresses are the same, situation that should not happen in practice. While in the previous example we hard-coded a value (Line 14), in this case we use variable binding (Lines 18,19).

In order to be more expressive, the function name is given by the atom 'Is it the same?'. The *side-effect free* **is_binary** function is used to check whether the provided argument is of binary type.

The **binary_to_list** function converts the binary argument to a list of integers.

The format for specifying a number in a given base is Base#Number. Please note the lower-case hexadecimal representation ffffff (Line 9) of the broadcast address. If upper-case letters would have been used, then this would have been interpreted as a variable.

[6] http://www.erlang.org/doc/apps/snmp/index.html

[7] Usually a string is an array of chars usually stored on 2 bytes, but since in Erlang strings are lists of integer and an integer might need 4 or 8 bytes in x64 architectures, memory consumption is higher.

Listing 3. Machine

```
-module(machine).
%Public functions
-export(['What are the Erlang Runtime Statistics'/0,
         find_accessed_files/2,netstat_out/0,
         ps_out/0, ps_out/1,secret/0,set_secret/1,test/1]).

%Retrieve Erlang Statistics
'What are the Erlang Runtime Statistics'()-> statistics(runtime).
%Retrieve shared secret cookie
secret()-> erlang:get_cookie().
%Ser secret cookie
set_secret(Secret) when is_atom(Secret)->
         erlang:set_cookie(node(),Secret).

%Retrieve a list of file names from Directory
find_accessed_files(Directory,DaysBefore)
         when is_list(Directory)
          and is_number(DaysBefore)->
         ( (find_accessed_files_fun())
                  (Directory))(DaysBefore).

%Parse output of the netstat command
netstat_out()->
  'Parse command output disregarding first line'("netstat -i ").
%Parse output of  “ps elf” on Linux and display the output as a
%list of tuples
ps_out()->ps_out(" elf").

%Test functions
test(recent_changes)->
        ?MODULE:find_accessed_files("/home/marian",1);
test(ps)->?MODULE:ps_out();
test(netstat)->?MODULE:netstat_out().

%% Private functions, that are not visible outside the module
find_accessed_files_fun()->
    fun(Directory)->
        fun (DaysBefore)->
            (split)( (os_cmd)
                    ((string_print)
("find ~p -atime ~p -print", [Directory,DaysBefore])),"\n")
        end
    end.
remove_empty(ListOfLists)->
         lists:filter(fun(List)-> List=/=[] end,
                  ListOfLists).
ps_out(Options)->
         parse_cmd_out("ps "++Options).

'Parse command output disregarding first line'(Cmd)->
    case catch (tl)((split)((os:cmd)(Cmd),"\n")) of
      {'EXIT',_} -> {kernel_interface,
                    'Cannot get interface data'};
      Tail -> {kernel_interface,parse_line_by_line(Tail)}
    end.

parse_cmd_out(Cmd)->
         (parse_line_by_line)(((split)((os:cmd)(Cmd),"\n"))).

parse_line_by_line([HeaderRow|Rows])->
  Header = (remove_empty)((split)(HeaderRow," +")),
  lists:map(fun(Row)->
    (lists:zip) (Header,(remove_empty)((split)
         (Row," +",length(Header))))
         end,Rows).

string_print(Format,Argument) when is_list(Argument)->
         (lists:flatten)((io_lib:format)(Format,Argument)).
%Use regular expressions to extract data
split(Text,Exp)->
         re:split(Text,Exp,[{return,list},trim]).
split(Text,Exp,Parts)-> re:split(Text,Exp,[{return,list},
                                    trim,{parts,Parts}]).
%Execute  OS command or run external program and fetch
%output
os_cmd(Command)->os:cmd(Command).
```

Listing 4. Bit Syntax

```
1.  -module(bit_syntax).
2.  %Extract the Ethernet frame and format it as a tuple
3.  extract_ethernet_frame_data(
4.      <<MAC_Destination:6/binary, %6 bytes = 48 bits
5.        MAC_Source:6/binary,
6.        Type:2/binary,
7.        Rest/binary>>) %Rest is what remains, including CRC
8.      when is_binary(B)->
9.                  {MAC_Destination,
10.                  MAC_Source,
11.                 Type,
12.                 Rest}.
13. flooding(<<_:6/binary,
14.            16#ffffff:6/binary,
15.           _/binary>>=B
16.         )when is_binary(B)-> true;
17. flooding(X) when is_binary(X) -> false.

18. 'Is it the same?'(<<Same:6/binary,
19.                    Same:6/binary,
20.                    _/binary>>=B
21.                  ) when is_binary(B)->true;
22. 'Is it the same?'(B) when is_binary(B)->false.
23. binarytolist(D) when is_binary(D) -> binary_to_list(D).
24. data()-> <<0,23,154,219,234,116,0,29,9,96,222,135,8,0,
    69,0,0,52,87,19,64,0,64,6,129,249,192,168,1,3,152,46,7,222,20
    2,243,0,80,80,215,70,118,54,56,116,241,128,17,0,69,13,32,0,0,
    1,1,8,10,2,69,134,183,48,133,64,99>>.
```

Listing 5. Sample run

```
# erl -sname e2@mercury

1> c(bit_syntax,[debug_info]). %compile module
{ok, bit_syntax}
2> D = bit_syntax:data().
<<0,23,154,219,234,116,0,29,9,96,222,135,8,0,69,0,0,52,87,
  19,64,0,64,6,129,249,192,168,1,...>>
3> bit_syntax:extract_ethernet_frame_data(D).
{<<0,23,154,219,234,116>>,
 <<0,29,9,96,222,135>>,
 <<8,0>>,
 <<69,0,0,52,87,19,64,0,64,6,129,249,192,168,1,3,152,46,7,
   222,202,243,0,80,80,...>>}
4> bit_syntax:flooding(D).
 false
5> bit_syntax:'Is it the same?'(D).
 false
```

Listing 6. Parsing

```
{blocked,tcp,
         {source,{ip,79,136,116,120},{port,20867}},
         {destination,{ip,188,24,20,60},{port,20097}}
}
```

E. Data pre-processing

Depending on the observations type, whether they are application output, **syslog** messages or packets sniffed from the local network, different strategies can be adopted among which *regular expressions* and the *bit syntax*.

In the case of **syslog** messages, we extract information about dropped packets by using regular expressions on the data in Listing 2, in a similar manner to that in Listing 3 to obtain the information presented in Listing 6.

In the case of sniffed network packets, the bit syntax makes it easy to extract features from a Protocol Data Unit as previously shown. Furthermore, with the help of **bistring** we have access at **bit level**, not only at byte level.

This enables us to perform noise filtering and / or

normalize the data we have collected in an easy manner.

1) Noise Filtering

The observations we make affect the system whose state we want to observe. Let's consider a *centralized* architecture in which sensors placed on multiple hosts send observations to one or more Erlang nodes for processing. For doing this, the observations must be encoded as a TCP/IP Protocol Data Unit, or simply packet, and carried through the network. This involves a number of system calls, which increase, for example, the number of processor interrupts and context switches.

We consider this to be noise, and we must filter it, either by using noise tolerant algorithms or by adding supplementary resources. In the later case, one obvious solution is adding a dedicated network interfaces for sending observations, but this is not a scalable solution. Another strategy is to combine host based intrusion detection with network based intrusion detection [14].

With Erlang this can be easily achieved. Sniffed network packets can be filtered using the bit syntax, meaningful data from OS's logs or **syslog** messages can be extracted and outliers identified. Furthermore, we can query the system to see what the impact of the IDS on the system is.

2) Normalization

A common issue related to gathering observations is that they depend on the underlying **context** [15]. Anomaly based IDS learn the context and then establish whether a set of observation values are a sign of an attack, or not.

In the case of a *distributed approach* [16] in which multiple hosts collaborate in detecting an attack, problems arise from *how much* and *what* data to share.

If the host based values are aggregated into categorical or real valued observations, they might be of limited utility for neighboring nodes. If they have utility, then we expect them to increase the host's processing load.

Sharing the attack data can cause security problems and increase the network traffic, but at the same time it could allow a host to learn the behavior of its peers. Comparing its own behavior to that data and taking into consideration its peers behavior, it might detect if the peer was compromised by an attacker. But, in the absence of a reference host, establishing what is normal for multiple hosts is a difficult task. The obvious solution is to have a group of nodes (Erlang nodes) that would process observations from all the hosts, and decide if an individual host or the network is under attack.

F. Dimensionality reduction

In the case of sniffed network packets, the bit syntax makes it easy to extract features from a *Protocol Data Unit* (**PDU**). In Listing 4, we use binary pattern matching to check the source and destination addresses. While an FF.FF.FF.FF.FF (in hexadecimal) is a valid destination address (the broadcast address), it is not a valid source address: its purpose is to flood the local network.

Problems also arise with the data itself. Let's consider multiple identical **syslog** messages, from the same host. Processing them requires resources, so we might take the first one, and disregard the rest. Doing so, we lose information: a message that appears 10 times might be more important than the same message occurring only once.

To illustrate this, let's take **syslog** messages related to invalid user logins: it might happen for a user to wrongly type its password one time, but when this happens 10 times this is a sign of an intrusion attempt.

In the case of host *based intrusion detection*, if we augment the message with additional data to express the number of times it appeared, state should be stored across reboots. A malicious user might, or the OS would, reboot the system after a number of invalid attempts, to mask the intrusion attempt in the first case or to protect itself in the later. When hosts send **syslog** login messages to a processing node a stateful approach is required to keep a count of login attempts.

1) Feature selection

Features should be selected such that the difference between attack and normal observations should be significant in order to facilitate the clustering / classification. The attack vs. normal difference's extremes range from 1 bit to as large as the whole packet payload. The first case, happens when the attacker spoofs IP addresses, or not, and sends packets with the TCP RST (reset) bit flag set to cause the tear-down of the network connection between the sender and the intended receiver, while the second case is common when in the middle of a TCP connection establishment (SYN=1, ACK=0) a large payload is present[8] to cause a buffer overflow. These extreme cases are old and have been addressed, but they help to keep things into perspective.

To overcome such issues, in a centralized approach multiple hosts send messages to a single Erlang node. In turn, this will collect all the observations and correlate them to see whether an underlying exists. Starting from the raw data, a search is performed to see what features are significant and what group of features best describes the state of the system.

Population based genetic algorithms can be easily implemented due to the concurrency and the communication facilities offered by Erlang.

Furthermore, one could create and use Erlang expressions at run-time, using the **erl_scan:string**, **erl_parse:parse_exprs**, **erl_eval:add_binding** and **erl_eval:exprs** functions.

2) Feature projection

Reducing a set of features to a single real value and using a threshold to decide whether an attack is undergoing is a common strategy in Intrusion Detection. **Artificial Immune Systems** and **Novelty detection algorithms [3]** are such an example. The principle is to label new packets as self or non-self based on their similarity with previously encounter packets under normal operation. This requires building a model of network packets (either inbound or outbound traffic, or both).

Another interesting strategy is the use of **Bloom filters**. They are a memory efficient way for checking if a feature value has been previously encountered [18]. Such a filter uses many hash functions for transforming a feature or a group of features into a single integer that is an index in a

[8] As an example the RFC 793, section 3.4 requires data in SYN packages and must be kept until the connection is established [17].

bit string. By setting the corresponding bit on, the bit string becomes a memory for the presented features and it is used for testing whether a pattern has been encountered or not.

G. Prediction

Due to the offered concurrency support and process encapsulation, algorithms can *simultaneously* run on the same, or on different, input data and using past performance statistics decide which classifier guesses better or what clustering algorithm has best tolerance to noise.

K-means and K-medoid help in building clusters on which more expensive classification algorithms would be applied. As an illustrative example, let's consider that we have 10 clusters. In the training phase, for each cluster we evaluate a set of classification algorithms and choose, for each cluster, the best one. When a packet is present, the clustering algorithm tells what it resemblance to and the classification algorithm tells what it thinks it is: normal or attack.

Although we perform redundant work, the classification task should improve since inside the designated cluster we expect the differences to be small and significant, and the amount of noise to be reduced.

The examples in a cluster can also be used for building a Bloom filter, which will be later used for establishing the membership of examples to a cluster.

The development of classification algorithms is further facilitated by the distributed nature of Erlang. Interesting example for this are Artificial Immune Systems for Intrusion Detection. Erlang has features making it suitable for building such a system: parallelism, message passing communication, lightweight processes, and the hibernation concept.

In the following, we provide some guidelines of how AIS can be implemented in Erlang. The AIS's cell can be viewed as regular Erlang process. Such cell processes can be spawned in large numbers on any number of nodes in the distributed system. The exchange of information between cells is done by message passing. When a cell process does not receive a message in a long time, it would require to be put in the hibernation state, to consume less system resources. When a message is received, the cell is awakened and processes the message on the basis of its previous knowledge.

Since Erlang provides a whole platform for failure recovery, a parallel can be made to AIS's robustness [3]. For example, if a cell process in a supervision tree encounters an unexpected death due to failure it would be re-spawned.

H. Creating test data

Evaluating IDS requires gathering test and training data, which is challenging since labeled data is hard to find. A way to overcome this issue is to use existing software and hardware resources. Let's assume that we have sniffed some packets from the local network and that we have a commercial hardware or a software firewall, capable of sending **syslog** messages when a packet is dropped due to the violation of a security policy. With the application presented in Listing 1 we can address the problem of labeling training data, a task previously accomplished by experts [19]. Due to the scarcity of label data and to the fact that manual labeling is neither cost, nor time effective, this strategy is rather useful.

III. CONCLUSIONS

In this paper we have shown that the Erlang programming language can be used for developing IDS that exhibit robustness and scalability. Erlang's features such as hot code loading allow updating the IDS without affecting the current operation of the system. The supervision tree allows failed components to be restarted. Concurrent distributed processing is another feature useful when using AIS as a basis for IDS. While the processing power of an Erlang node is smaller than that of C based firewall, the Erlang system takes better advantage of multi-core architectures and provides easier application development due to its functional nature.

REFERENCES

[1] J. Greensmith, J. Feyereisl, and U. Aickelin, "The DCA: Some comparison – a comparative study between two biologically inspired algorithms," Evolutionary Intelligence, vol. 1, pp. 85–112, 2008.

[2] S. X. Wu and W. Banzhaf, "The use of computational intelligence in intrusion detection systems: A review," Applied Soft Computing, vol. 10, pp. 1–35, 2010.

[3] J. Twycross and U. Aickelin, "Information fusion in the immune system," Information Fusion, vol. 11, pp.35–44, 2010.

[4] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," ACM Comput. Surv. vol. 41, no.3, pp. 1–58, 2009.

[5] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[6] R. Virding, C. Winkstrom, and M. Williams, "Concurrent Programming in Erlang," J. Armstrong, Ed. Prentice-Hall, 1996.

[7] R. Carlsson, "Parameterized modules in Erlang," in ERLANG 03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang. New York,NY, USA: ACM, 2003, pp. 29–35.

[8] J. Armstrong, "Programming Erlang: Software for a Concurrent World." The Pragmatic Bookshelf, 2007.

[9] J. Nystrom, P. Trinder, and D. King, "Evaluating high-level distributed language constructs", in ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, New York, NY, USA: ACM, 2007, pp. 203–212.

[10] "Erlang Run-Time System Application", 5th ed., Ericsson, Nov. 2009.

[11] V. Nicosia. (2007, Oct.) "Towards hard real–time erlang." ACM SIGPLAN Erlang Workshop ICFP 2007.

[12] M. L. Bailey, B. Gopal, M. A. Pagels, and L. L. Peterson, "PATHFINDER: A pattern-based packet classifier," Proceedings of the First Symposium on Operating Systems Design and Implementation, USENIX, 1994.

[13] E. Nakashima, "Cyber attack data-sharing is lacking, congress told," The Washington Post, September 2009.

[14] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner, "State of the practice of intrusion detection technologies," Carnegie Mellon, Software Engineering Institute, Tech. Rep., 2000.

[15] S. Preda, F. Cuppens, N. Cuppens-Boulahia, J. G.Alfaro, L. Toutain, and Y. Elrakaiby, "Semantic context aware security policy deployment," in ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. New York, NY, USA: ACM, 2009, pp. 251–261.

[16] A. Abrahama, R. Jainb, J. Thomasc, and S. Y. Hana, "D-SCIDS: Distributed soft computing intrusion detection system", Journal of Network and Computer Applications, 2007.

[17] DARPA, " Rfc793 - transmission control protocol", sep 1981.

[18] 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.

[19] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman, "An overview of issues in testing intrusion detection systems," National Institute of Standards and Technology, Massachusetts Institute of Technology Lincoln Laboratory, Tech. Rep.