Indice

1 INTRODUCTION **2**

2 IDS INTERNAL STRUCTURE **3**
2.1 idsEngine child 3
2.2 logInit 6

3 IDS CONTROL SCRIPT **7**

4 GRAPHIC INTERFACE **7**

5 FUTURE WORKS **10**
5.1 Run-time decision context 11
5.2 Defense layer 2 11

1 INTRODUCTION

Why create an ids written in Erlang? Because an ids should be active 24/7 and resistant to failures. This is achieved thanks to the intrinsic features provided by BEAM (hot code reload and fault tolerance through supervisor processes and worker processes). Through this document I am going to explain the internal and external functioning of the nids, dividing it into 5 chapters listed below:

1. Internal structure of the program (otp tree explained).
2. Explanation of nids startup script.
3. Graphics window.
4. Future works.

2 IDS INTERNAL STRUCTURE

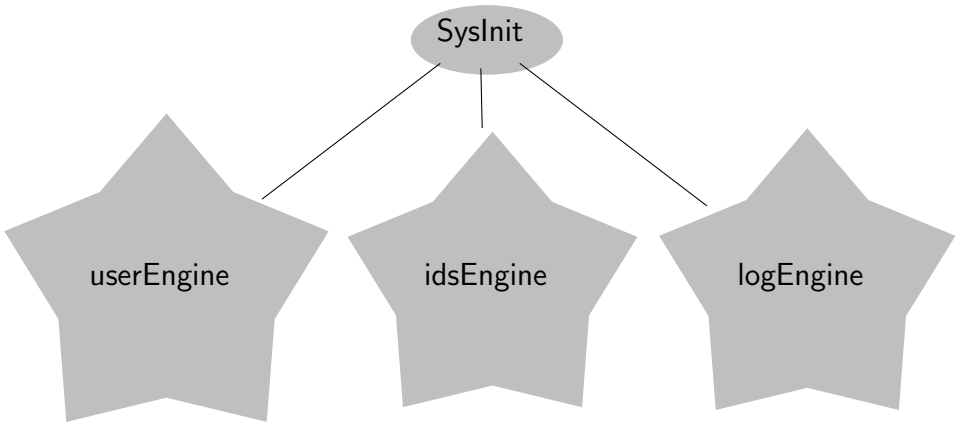

Figura 1: General structure of the ids

This chapter gives a general overview of the structure and internal functioning of the IDS, without going into the detail of the code. The folder structure of the IDS code is structured similar to the otp tree that will be shown in this chapter. The whole application is structured as an otp tree, made up of supervisors and workers. An engine is understood as an otp subtree that performs a certain function and can itself be composed of other engines. As you can see from the figure 1, the IDS is made up of 3 main engines. The first child of *sysInit* (userEngine) takes care of the graphics window for the user and general user-interaction. The second child of *sysInit (idsEngine)* consists of the actual ids engine. Finally we have the third child *logEngine*, which takes care of initializing the various handler modules for logging by the IDS. I place emphasis on *idsEngine* and *logEngine*, as *userEngine* will be covered in more detail in Chapter 3.

2.1 idsEngine child

As can be seen from 2, *idsEngine* is itself divided into two sub-engines, implemented by the *netDetector* subtree and the *netDefense* subtree respectively. *NetDetector* deals with the capture of incoming packets, passing from the construction of the flows, up to the classification of the flows obtained.

NetDefense instead deals with the "reaction" to the attack flows recognized by the IDS. For now *netDefense* 3 core is composed of the *defenseLayer1* agent, whose function, received in input a malicious stream, is to notify the user of the attack by providing him with information about it (ex: destination port etc ...), and in the case of port scanning attack, notifies you of the

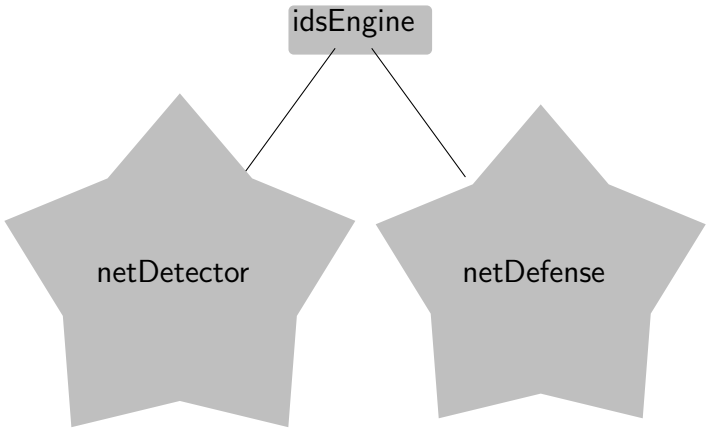

Figura 2: Ids core engine

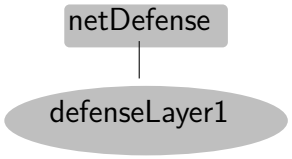

Figura 3: Net defense engine

UDP and TCP ports open on the host. Instead *netDetector*, as introduced before, turns out to be more complex, as it deals with more features. As you can see from the figure 4, it is further subdivided into the three sub-engines, each of which performs its own functionality but connected with the others.

Starting from the left, we find the *netHandler* subtree. It is responsible for the capture of packets and the construction of flows, "agglomerating" the captured packets according to a univocal identification [1] of the flow, obtainable from the properties of a packet (ex: ip source etc ..). *NetHandler*, show in the figure 5 is composed of the two sub-trees *sniffer*$Watcher$ and *flowBuilder* which, respectively, take care of collecting and processing the packets, to create the various flows to be analyzed. *Sniffer*$Watcher$ basically implements the ISO / OSI stack, where each layer corresponds a process, creating a **packet** record containing all the information about it (ex: source ip, protocol level 4, etc.). *FlowBuilder* consists of two processes: *netFlow* and *flowsStorage*. *NetFlow*, having received a packet, dispatches it to the correct *FlowRecorder* process or create the related *FlowRecord* process, based on the packet id. *FlowRecorder* builds and maintains one-way flow until it exhales [1]. After a stream has expired, *FlowRecorder* sends it to *FlowStorage*, which takes care of merging two complementary one-way streams, creating the entire stream. Two one-way flows are complementary if they

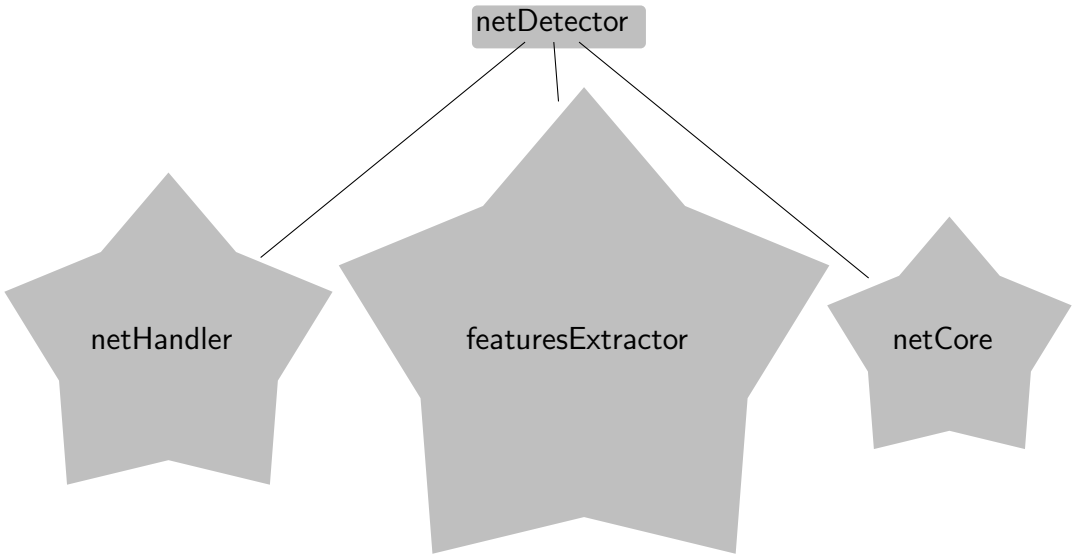

Figura 4: Net detector engine

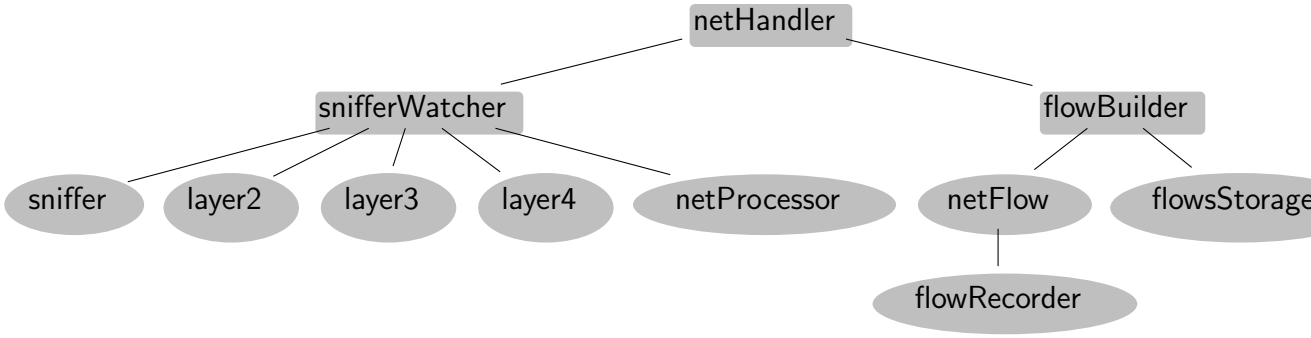

Figura 5: NetHandler engine

have their respective dual ids.
Example: **idFlow1** {ipsrc = 1.1.1.1 ipdst: 2.2.2.2 portadst= 44 portasrc= 55 protoTrans= tcp protoService: http}
dualIdFlow1 {ipsrc = 2.2.2.2 ipdst = 1.1.1.1 portsdst = 55 portadst = 44 protoTrans = tcp protoService = http }

The second child and component of *netDetector* is *featureExtractor* 6. *FeatureExtractor* is responsible for extracting the features of the dataset given a bidirectional flow sent by *flowsStorage*, creating a **features record** from the flow. Finally, the third child component of *netDetector* is *netCore* 7. *NetCore* ,as the name implies, it is the core of the stream analytics engine. It is composed of *erlDecisor*, which takes care of classifying the flow and training the classifier, in accordance with the features and targets of the

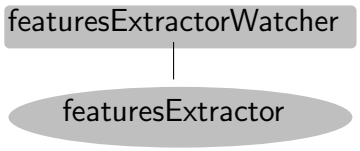

Figura 6: Engine of features extraction

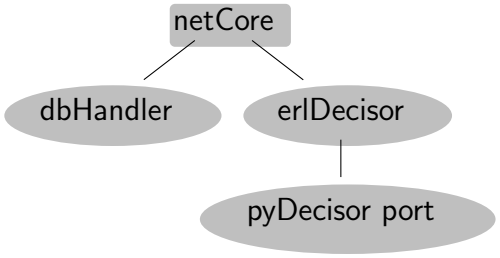

Figura 7: NetCore engine

dataset, using the python port *pyDecisor* that actually implements the classifier. After the classification of the **features record**, *erlDecisor* sends the **classified record** to *defenseLayer1* and possibly to *dbHandler* if enabled by the options (see chapter 3). *DbHandler*, for its part, is responsible for managing the data set. At present its functions are to pass the dataset to *erlDecisor* for training, create a new dataset and possibly save a classified features record.

2.2 logInit

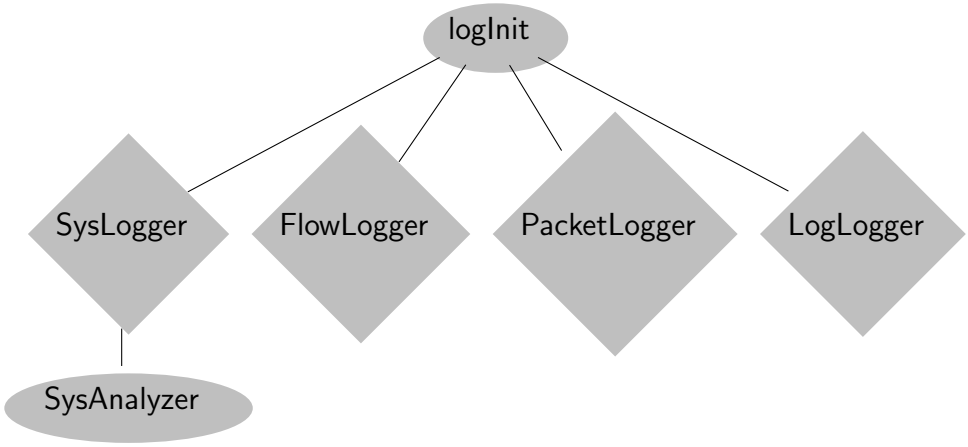

Figura 8: IDS logging structure

LogInit implements the IDS logging component 8. More precisely, 4 handler modules are implemented for the OTP logging API. *SysLogger* takes care of recording on file the errors at run-time of the system (with the relative information), typically crashes of workers or supervisors. As you can see, 1textitsysLogger communicates with the *sysAnalyzer* worker. *SysAnalyzer* implements a "global supervision" mechanism, ie all the error logs sent by *sysLogger* are stored by *sysAnalyzer* and once a certain number of **errors threshold** is exceeded, the **system condition** is calculated on the basis of the average time of the intervals between the various errors. If this condition is less than a minimum value then the system is shut down. The analysis of the system after reaching the error threshold is logged. *FlowLogger* and *packetLogger* take care of recording, respectively, the information on a flow **(id, info, classification)** and the various packets (record packet) in transit in the system. Finally *logLogger* takes care of recording failures in logging, it is a kind of metalogger.

3 IDS CONTROL SCRIPT

As anticipated in the README, the nids script allows a basic management of the ids, such as switching on / off, etc. By performing a basic management of the IDS, the set of commands it supports is itself basic, supporting only the main 5 that I am going to explain below.

1. **Compile**: recompile the whole ids source code.
   command: *nids compile*
2. **Start**: Start the ids, in case it is not already logged on.
   command: *nids start*
3. **Stop**: Stop the ids, in case it is active.
   command: *nids stop*
4. **opt**: Gets the GUI of the IDS .
   command: *nids opt*
5. **config 'cookieArg' 'nidsNodeNameArg'**: change the cookie and node name configurations of the nids.

4 GRAPHIC INTERFACE

The graphic interface of the IDS is divided into three tabs where each deals with a particular aspect of the functioning of the IDS.

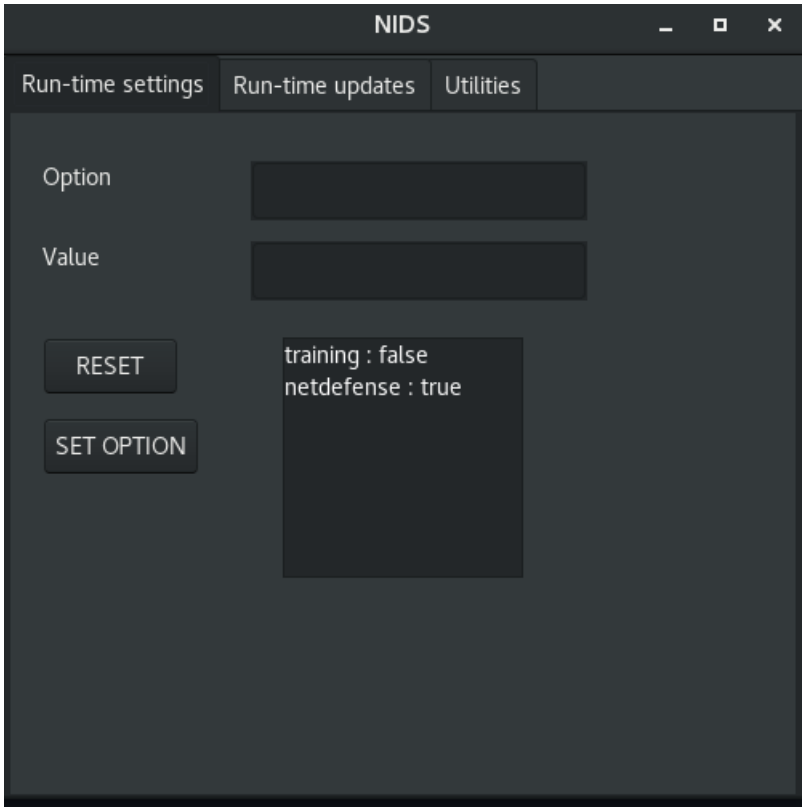

Figura 9: IDS options setting window

In the figure 9, the IDS options tab is shown. By setting these options you change the behavior of the ids in different points of the execution flow (NB: the execution flow must be defined). Currently only two options are supported:

- **Training**: if set to true, the records of the features classified by *erlDecisor* and sent to *dbHandler* (see chapter 2), are stored in the internal IDS dataset, to be used for future training of the classifier.
- **NetDefense**: if set to true, the *netDefense* engine is activated

The default values are false for training and true for netDefense. By inserting the option name with its Boolean (true or false) in the respective textboxes and pressing on "SET OPTION", you change a system setting.

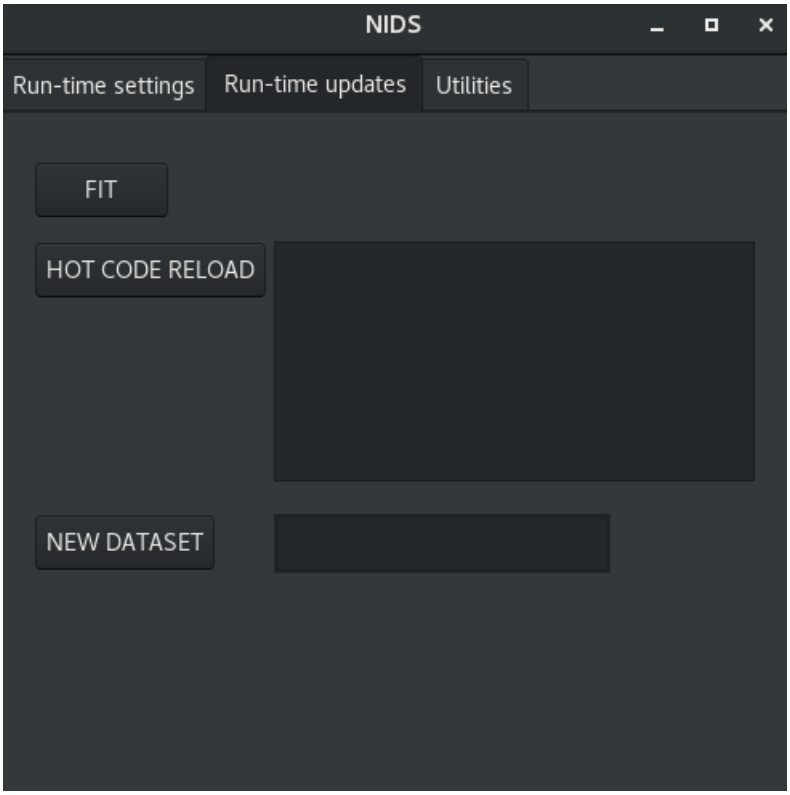

Figura 10: IDS internal structure setting window

In the figure 10, the IDS internal core tab is shown. Foreach phase connected to its relative button, it is strongly advised not to turn off the IDS during this phase.

- **Fit** button realigns the classifier on the internal IDS dataset. This operation is blocking that is until it ends, the user interface is disabled. Also, during retraining, some points of the execution flow are suspended, so as not to overload the system.
- **Hot code reload** button it allows the hot code reload.
- **New Dataset** button given a path containing at least one csv, create a dataset with the relative metadata files (see README).

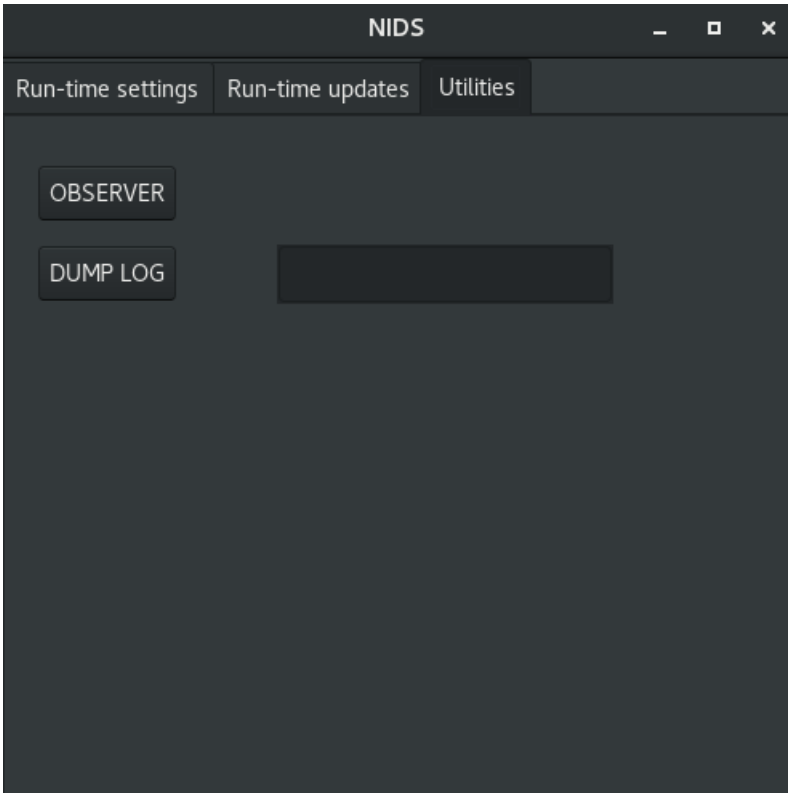

Figura 11: IDS utilities

Finally we reach the last tab.

- **Observer** button, start an *observer* erlang utils which shows the internal status of the ids node.
- **Dump log** button, given a path, clean and export in the inserted path the logs file of the *logEngine* described in chapter 2.

5 FUTURE WORKS

List below 3 features to be implemented in the future:

1. Run-time change of the ids decision context.
2. Structuring the ids as an otp application [2].
3. Implement the *defenseLayer2*.

Below I explain in more detail the points 1 and 3, as on 2 there is the link on the bibliography.

5.1 Run-time decision context

AIn the state of the art, the only significant change at run-time is the change in the decision context of the IDS. With run-time change of decision context i mean the radical change of the type of dataset, that is, features and classification targets. Obviously the new dataset should always be inherent to the classification of network flows. In part this functionality is implemented thanks to the functionalities introduced in the previous chapters. Below I show a pseudo-algorithm that implements this functionality:

1. Stop packet capture and build streams: this results in a stop of operation followed by a global flush of the various processes involved, therefore *idsEngine*.

2. Build the new dataset: This step is already implemented thanks to the **New Dataset** functionality introduced in chapter 3.

3. Change the function createRecordFeatures in *featureExtractor* child.

4. Reload the new code using the **Hot code reaload** functionality introduced in chapter 3.

5. Create new model and retrain it on the newly created dataset: this step is already implemented thanks the **Fit** functionality introduced in chapter 3.

6. Restart the entire *idsEngine* stopped previously.

As you can see, most parts of the algorithm, in details points 2,3,4 and 5, are already implemented.

5.2 Defense layer 2

As explained in chapter 1, *netDefense* engine is only equipped with *DefenseLayer1* worker which takes care of notifying the user of a potential attack. So currently only passive protection would be offered. The idea behind *DefenseLayer2* is to implement an active defense mechanism by the IDS. This results in the manipulation of the network flows arriving on the host through the manipulation of the firewall present on the host (on linux there is nftables as successor of iptables). Currently a mere manipulation of the rules of nftables is already present: [3]. Is a simple nftables command wrapping library, also very recent and, obviously, to be tested in more depth. But the real challenge is to implement an automatic and intelligent manipulation of

these rules, through the aforementioned library, based on the attack notifications coming from *DefenseLayer1*. This functionality could be obtained through another machine learning mechanism, similarly to what happens for the classification of flows.

Riferimenti bibliografici

[1] Anna Sperotto. "Flow-based intrusion detection". Tesi di dott. University of Twente, Enschede, Netherlands, 2010. URL: http://eprints.eemcs.utwente.nl/18649/.

[2] Erlang Official Documentation. *Erlang Otp Application*. Rapp. tecn. URL: https://erlang.org/doc/design_principles/applications.html.

[3] Luca Tabanelli. *Nftables erlang API*. Rapp. tecn. URL: https://github.com/Taba92/nftables_erlang_api.