



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Big Data for Digital Forensics

MASTER DEGREE: Master in Science in Telecommunication Engineering
& Management

AUTHOR: Alfredo Daniel Cuzcano Cossi

ADVISOR: Juan Hernández Serrano

DATE: February, 15th 2018

Title: Big Data for Digital Forensics

Author: Alfredo Daniel Cuzcano Cossi

Advisor: Juan Hernández Serrano

Date: February 15th, 2018

Abstract

Digital Forensics and its sub-branch Network Forensics are important and relevant topics which have gained further attention with the DDoS attacks delivered by botnets.

This work focuses on a novel IDS solution called: SLIPS. This is a free software that uses Machine Learning to detect malicious behaviors in a network with the use of Markov Chain based detection and previously trained models. A major limitation of SLIPS lies on its performance, and this work also touches on the topic of Big Data, and more specifically MapReduce, in order to aid SLIPS with a better resource utilization.

With the redistribution of SLIPS tasks across workers, adding a pre-processing of data, the proposed solution using MapReduce presented performance improvements of up to 433 times with the datasets tested.

ACKNOWLEDGEMENT

First of all, I would like to thank my advisor, Juan Hernández, for his patience and guidance throughout this work. You pointed me in the right direction for this work.

Also, I would like to thank my colleagues and professors of the MASTREAM program, I have learned so much from you, I cannot put it into words.

I would like to dedicate this work to my parents, for their love, support, guidance and encouragement. And to my love, Mauranne, without your caring, support and lessons, I wouldn't be here writing this words, thanks, right now and forever.

And finally, I would like to thank Sydney Banks, without the understanding he shared throughout his life, this work wouldn't have been possible. In your own words:

"The answer to all complexity lies in simplicity."

CONTENTS

INTRODUCTION	1
CHAPTER 1. BACKGROUND	2
1.1. Digital Forensics	2
1.1.1. Definition of Digital Forensics	2
1.1.2. Digital Forensics Branches	2
1.2. Network Forensics	3
1.2.1. Definition of Network Forensics	3
1.2.2. Types of Attacks in Network Forensics	3
1.2.3. Types of Attacks: Botnets	4
1.3. IDS for Network Forensics	6
1.3.1. Definition of IDS	6
1.3.2. IDS and Network Forensics	6
1.3.3. Detection Styles of IDS	6
1.4. Examples of IDS	7
1.4.1. Snort	7
1.4.2. Stratosphere IPS	7
1.5. Challenges of IDS for Network Forensics	7
1.6. Evolution of Data Analytics for IDS	8
1.7. New solutions for New challenges	9
1.7.1. Big Data	9
1.7.2. Types of Analytics for Network Forensics	10
1.7.3. Big Data Tools and Techniques in Network Forensics for IDS	11
1.7.4. Usage examples	13
CHAPTER 2. STRATOSPHERE IPS	15
2.1. Motivation	15
2.2. Description of the Stratosphere Project	15
2.3. Description of SLIPS	15
2.4. Description of Datasets	16
2.5. Implementation details of SLIPS	16
2.5.1. Description of SLIPS processes	16
2.5.2. Representation of patterns in the network	19
2.5.3. Computing the patterns as letters	19
2.5.4. Description of the Machine Learning: Markov Chain	21
2.6. Limitations of current version of SLIPS	22
2.6.1. Degradation of performance	22
2.6.2. Time windows usage in source code	23
2.6.3. Time windows size and convolutional networks	23

2.6.4. Redundancy of periodicity in letters and symbols	24
2.6.5. Documentation of thresholds for SDW	24

CHAPTER 3. PROPOSAL FOR STRATOSPHERE LINUX IPS..... 25

3.1. Motivation..... 25

3.2. Description of proposal 25

3.3. Description of package used in proposal 25

3.3.1. Proxy Objects 25

3.3.2. Pool..... 26

3.3.3. Cpu_count 26

3.4. Implementation of Proposal 26

3.4.1. Main Process 26

3.4.2. Function: MapDetection..... 29

3.4.3. Function: ReduceDetectionResults 31

3.5. Comparison of Tests between Proposal and SLIPS 32

CONCLUSIONS..... 35

ACRONYMS 37

REFERENCES..... 38

ANNEX A 42

INTRODUCTION

Digital Forensics and its sub-branch Network Forensics are important and relevant topics. These topics are diverse, with different types of attacks and between all of them, one has gained further and further attention: the Distributed Denial of Service (DDoS) attacks delivered by botnets.

Related to Network Forensics, we find the Intrusion Detection System (IDS) solutions, based on signature-based and heuristic approaches. Threats are evolving and IDS together with them, and new challenges have appeared, mostly the vast amount of data presented in networks.

This work focuses on a novel IDS solution called SLIPS [1]. This is a free software part of the Stratosphere project [2] that uses Machine Learning to detect malicious behaviors of the traffic related to Command and Control channels of botnets in a network with the use of Markov Chain based detection and previously trained models. In addition to the usage of Machine Learning, this model also utilizes real botnet captures, which adds relevance to its usage. A major limitation of SLIPS lies on its performance and its continuous degradation over time, most of the improvement on its code have been focused on accuracy improvement.

This work also touches on the topic of Big Data, introduces some previous work done on the topic and discusses some relevant technologies, more specifically MapReduce [3].

SLIPS works with two processes, a main one and a background one. But the tasks across them are not balanced. With the redistribution of SLIPS tasks across workers, adding a pre-processing of data, and taking from MapReduce the mapping of detection across different independent workers, the proposed solution using MapReduce aids SLIPS with a better resource utilization and presented performance improvements of up to 433 times faster with the datasets tested.

The remainder of this document is organized as follows. The first chapter presents some background topics. The second chapter focuses on the SLIPS solution, the description of its code and the workflow and a presentation of its limitations. The third chapter presents the proposal solution of this work.

CHAPTER 1. BACKGROUND

1.1. Digital Forensics

1.1.1. Definition of Digital Forensics

A regular and early definition of Digital Forensics is taken from the first Digital Forensics Research Workshop (DFRWS) in 2001 [4], with the intention of establishing it as a scientific discipline:

“Digital Forensic Science is the use of scientifically derived and proven methods towards the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations.”

A later definition in [5], indicates: “...is the practice of investigating computers, digital media, and digital communications for potential artifacts.”

When referring to artifacts, [5] indicates them as being any object of interest, since the word evidence refers to a court case; and highlights these artifacts as potential evidence.

In [6], a description of the principles of digital forensics are presented, which are the following and may help us have a better understanding of the subject:

- Previous validation of tools and procedures.
- Reliability of tools.
- Repeatability of processes.
- Documentation of procedures and processes (to allow repeatability).
- Preservation of evidence.

These principles are of importance, since as mentioned in [5], the product of this work can possibly have legal relevance in a court case.

These principles provide the basis for the workflow performed in Digital Forensics [6]:

- Identification of devices containing evidence
- Collection of devices
- Acquisition or producing an image of the potential evidence
- Preservation of evidence integrity
- Analysis of evidence acquired
- Reporting of results

1.1.2. Digital Forensics Branches

Digital Forensics divides itself into 5 branches or fields, as mentioned by [7]:

- Computer Forensics

- Network Forensics
- Mobile Device Forensic
- Memory Forensics
- Emails Forensics

We will present the definition for Network Forensics, which is in the scope of this work and the following chapters.

1.2. Network Forensics

1.2.1. Definition of Network Forensics

Within the scope of Digital Forensics, we find the sub-branch of Network Forensics [4] [7].

Also known as Digital Forensic Science in Networked Environments.

In [4], they provide us with an attempt of explaining it:

“The use of scientifically proven techniques to collect, fuse, identify, examine, correlate, analyze, and document digital evidence from multiple, actively processing and transmitting digital sources for the purpose of uncovering facts related to the planned intent, or measured success of unauthorized activities meant to disrupt, corrupt, and or compromise system components as well as providing information to assist in response to or recovery from these activities.”

In other words, as mentioned in [8], this field relates to the capture, recording and analysis of network events in order to discover incidents.

1.2.2. Types of Attacks in Network Forensics

In a network, we can find an attack in a variety of ways. We present here a summary of the main types of attacks, considering the literature of [5]:

- Denial of Service Attacks: It's a type of attack that makes a system or any other type of network unavailable to its intended users, sometimes called honest users. This attack is launched by a large number of distributed hosts, flooding the available resources. The most common subtypes are the following:
 - SYN Floods: For a TCP communication, an attacker sends the initial SYN and the victim responds with the SYN/ACK, leaving the connection in half-open state. The system, with its limited resources, awaits for the final ACK in order to establish the connection. The objective of a SYN flood is to simply fill up the slots that the target system has available for half-open connections. This is one of the most popular attacks nowadays [9].
 - Malformed Packets: Where the protocols have behaviors not in accord with their definitions (i.e. RFCs), and this present problems for the programs responsible for processing this input. An example of this is the Teardrop attack.

- UDP Floods: In this attack, because of the use of UDP the operating system is not doing the admission control that it does with TCP, so there is an increased amount of work for the processor. The attacker is looking to flood the processor by sending a large volume of UDP messages. This is an attack with the purpose of consuming all available bandwidth.
- Amplification Attacks: Here the attacker is looking for others to help him in his effort. An example of this, it's the Smurf attack. The Smurf attack relies on the spoofing of a source IP address and sending an ICMP echo request or ping message, to the broadcast address of a network block. If the network has not been correctly configured, every host on that network will receive an echo request, for which they will send an echo response back to the spoofed source.
- Distributed Attacks: It consists of multiple attackers distributed around the Internet, since multiple attackers in one site are going to be constrained by the amount of bandwidth available at the precise site. A distributed attack needs to be coordinated. These multiple attackers are known as a botnet. A botnet is a collection of bots, which are computes systems owned by someone else but under the control of the botnet owner or attacker.
- Vulnerability Exploits: in this case, network services allow attackers to send specially crafted messages that allows them to gain access to a system.
- Insider Threats: This is the case associated to attackers who where users or associated to users of an organization and gathered permissions for various resources on the network without ever dropping any of the permissions after leaving the position or the organization.

Due to the scope of this work presented in the following chapters, a more in depth explanation of the type of attacks involving botnets is detailed in the following section.

1.2.3. Types of Attacks: Botnets

Botnets are considered one of the most major threats on the Internet [10], capable of delivering Distributed Denial of Service (DDoS) attacks, or large-scale spam campaigns [11].

The word botnet is formed from the word 'robot' and 'network' [11]. These botnets are a network of infected hosts called bots, controlled by an attacker known as the botmaster. Botmasters sends commands via a Command and Control channel [10]. The Command and Control channels refer to the implementation of communication protocols that allow the owners or attackers to control all the infected computers, with the intention of synchronizing actions on them [12]. Among the actions performed by the bots, we can find port scanning, binary download, exploit attempts and SPAM sending [12].

Botnets differ themselves not on how they infect, but on how they spread: each infected host is part of an overlay network [12].

As of how they work, they originally relied on a central architecture, with a core network and a few interconnected IRC (Internet Relay Chat) server [10]. When the botmaster wanted to send any type of command, they were pushed to the IRC channel and all the bots receive it.

Since 2003, botnets are based on P2P (Peer to Peer) protocols, where each bot act as a client and a server [10] [12]. In this case, the bots ask the commands to their servers, in order words, the commands are being pulled. This present the following advantages in contrast to previous approaches [12]:

- They can be decentralized.
- They can hide the amount of bots infected.
- They can avoid being monitored or sniffed.
- They can be highly resilient to take down attempts.

Since 2005, these capabilities became enhanced, by starting to hide their traffic inside normal HTTP traffic in order to avoid detection [12].

To give us an idea of the size of these botnets, by February 2018 countries as India have approximately 2.5 million bots, and China 2 million bots, as reported in [13]. In Figure 1.1 by Kaspersky Lab, it shows us a comprehensive diagram of all the personal level and high-level consequences of botnets.

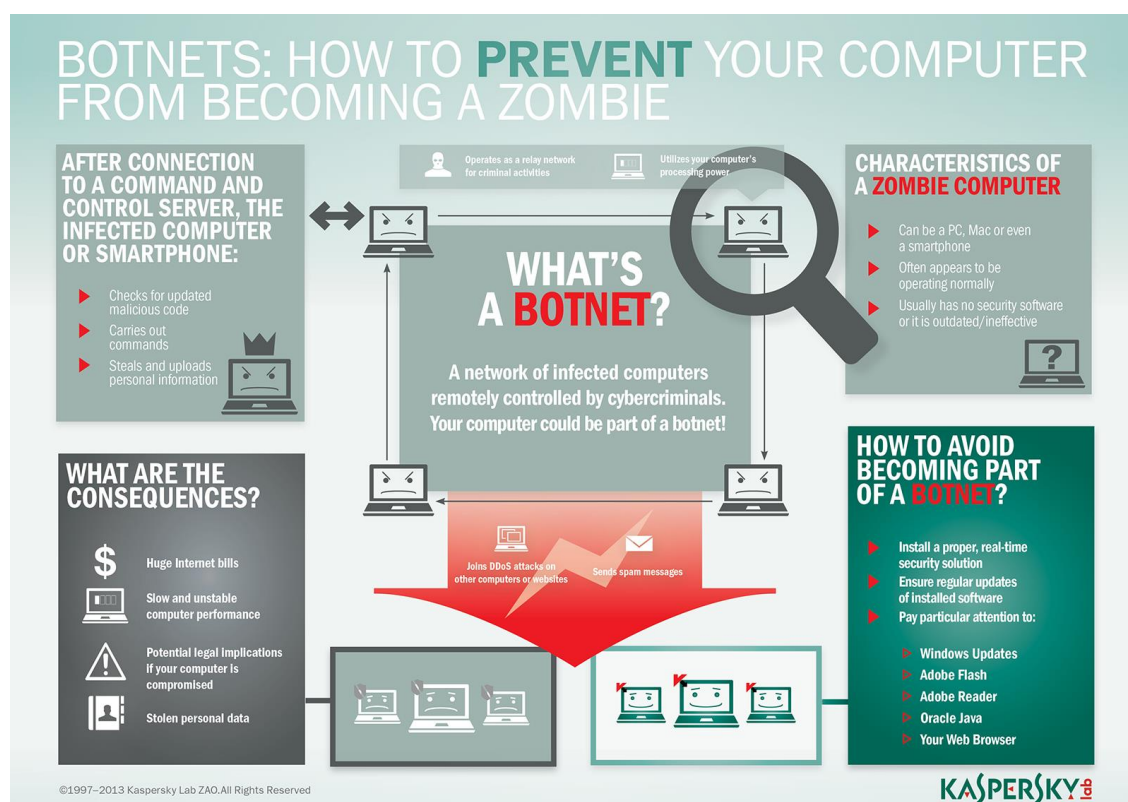


Fig 1.1 Botnets as presented by Kaspersky Lab [11]

1.3. IDS for Network Forensics

1.3.1. Definition of IDS

The concept of Intrusion Detection Systems (IDS) was initially presented in 1987 [14]. An IDS is a device that acquires information from a target information system to perform a diagnosis on its status. An overall goal is to discover breaches of security, attempts of breach or vulnerabilities to potential ones [15].

It can be seen as a detector that processes information coming from the target system, with the intention of protecting it. The target system monitored can be a workstation, a network element, a server, a mainframe, a firewall, a web server, or any other type of device [15].

Regarding the concept of Intrusion Prevention System (IPS), throughout the literature it is common to find the terms of IDS and IPS used interchangeably. They work similarly, but the difference lies in the action after the detection [16]:

- IDS detects intrusions and intrusion attempts; in addition to this, issues alarms, alerts and logs.
- IPS detects intrusions and intrusion attempts; in addition to this, it may block or prevent these activities in real-time.

IPS can be seen as an extended version of the IDS [16].

1.3.2. IDS and Network Forensics

In addition to identifying intrusions as they happen, IDS can also generate data that could be used during the course of a Forensic Investigation [5]. For this reason, it is important within the field of Network Forensics to understand the output and how an IDS works.

1.3.3. Detection Styles of IDS

IDS have different ways of determining if a traffic is infected as stated in [5]:

- Signature-Based: Identify a pattern in the malicious network traffic, this pattern is the signature. The limitation lies in the fact that the illicit action has to happen before the identification (signature) has been created. A network signature may be the source of the IP address, or a particular port used. Or it can also be related to a pattern of the network traffic as presented in [17]
- Heuristic: Also referred to as anomaly-based. It is based on the idea that we can determine what normal looks like, and anything that is not normal is considered as an anomaly [15].

1.4. Examples of IDS

Following there are two examples of IDS. Snort is a well-known IDS, whose work has been mentioned in [14] and [18]. Stratosphere IPS is a novel IDS which proposes a distinct way of detecting malicious behavior, and for such we consider it relevant in the following examples.

Another well-known IDS are Suricata and Bro, but they will not be presented in this work since their behaviors and functions are based on the one of Snort [5].

1.4.1. Snort

It is an open source network intrusion detection system, capable of performing real-time traffic analysis and packet logging on IP networks [19] [5], and created in 1999 by Martin Roesch, and bought by Cisco Systems in 2013 [20]. It can perform protocol analysis, content searching/matching, and it can also be used to detect a variety of attacks and network probing.

It has three primary use cases [19]:

- Packet sniffer.
- Packet logger.
- Network Intrusion Prevention System.

Snort is highly configurable, in addition to multiple options of output, it also has the ability to add modular plug-ins called preprocessors [5].

After the preprocessor, Snort utilize rules. They are based on types of attacks. A limitation is that Snort source does not come with a default set of rules, some Linux distribution will include them, or they need to be pulled for Snort.

1.4.2. Stratosphere IPS

Stratosphere IPS is a free software, presented as an IPS that uses Machine Learning to detect malicious behaviors in the network traffic [2]. It is presented as an IPS, nevertheless its current version doesn't allow for traffic blocking or any type of action besides the output alarms and logs [1], which are proper from an IDS (section 1.3.1).

Stratosphere IPS started in Python on Linux and Windows [21], and as communicated by their developers there is a version available that can be run on Snort as a plug-in based on the C programming language [2].

1.5. Challenges of IDS for Network Forensics

A few of the more remarkable challenges for IDS in Network Forensics are the following [22] [6]:

- Increased number of Sources: Evidence is becoming more and more heterogeneous. IDS faces now not only structured data, but unstructured

data, coming from distinct sources as removable media, mobile devices and cloud services.

- Increased in Network Traffic: The scale of Information nowadays has been increasing each year. Hackers can hide their presence easily these days, thanks to the increased amount of traffic daily between nodes.
- New technologies: New technologies are being developed, and with them, new challenges are appearing that Network Forensics didn't face 10 years ago:
 - Cloud Services: The emergence and development of Cloud Service providers, have made networks more vulnerable to new attacks.
 - IoT: Connectivity is becoming ubiquitous, and threats will increase with them.

One popular citation across the literature comes from a study done by IBM [23]: "90% of the data in the world today has been created in the last two years. This data comes from everywhere: sensors used to gather shopper information, posts to social media sites, digital pictures and videos, purchase transaction, and cell phone GPS signals to name a few. This data is big data."

1.6. Evolution of Data Analytics for IDS

The Cloud Security Alliance in [24], presented the evolution of Intrusion Detection in three parts:

- First Generation: The generation of Intrusion detection systems, this is where Security Architects realize the need for a layered security, since a 100% protective security is impossible. This can be seen as reactive security.
- Second Generation: The generation of Security information and Event management (SIEM), where these systems aggregate and filter the alarms from many sources for actionable data for the security analysts. The amount and frequency of logs used to be fixed to amounts like 60 days, due to limits in the underlying data technology [25].
- Third Generation: The generation of Big Data analytics in security. In this generation, we have the potential for a significant advance, since it will reduce the time for correlating, consolidating and contextualizing diverse security event information. Which will allow for correlating previously unused data, as long-term historical data. Security analytics will be transformed by:
 - Collecting data at a massive scale.
 - Perform deeper analytics.
 - Consolidate view.
 - Real-time analysis.

As an example, in [25] the case study of Zions Bancorporation was presented:

- A query among a month's load of data, took between 20 minutes to an hour.
- With Hadoop system, and the system running queries with Hive, they got the same results in about a minute.

1.7. New solutions for New challenges

Across the literature, we can find many solutions for tackling the challenges presented in the section 1.5, as part of the evolution suggested in the section 1.6.

First of all, some background definitions will be presented. Second, we will present some relevant tools and techniques discussed in some of the literature, and related examples of usage in a later section.

1.7.1. Big Data

When talking about Big Data, we are talking about data whose complexity hinders it from being managed, queried and analyzed through traditional storage architectures, algorithms and query mechanisms [22].

The complexity of Big Data is usually defined by the 3Vs (Volume, Variety, Velocity) [22], originally developed by Gartner analyst Doug Laney in 2001 [26]. A comparison of these 3Vs against traditional methods can be seen in Figure 1.2. We will present 3+2Vs, adding Veracity and Value to the mainstream definition [26], since with the evolution of Big Data, additional criteria should be taken into account:

- Volume: referring to the amount of data, from terabytes, petabytes or even more.
- Variety: referring to co-existence of unstructured, semi-structured and structured data.
- Velocity: referring to the rapid pace at which Big Data is being generated.
- Veracity: referring to the importance of maintaining quality data and on handling problems such as noise or missing values.
- Value: referring to the sense that if a particular data does not provide significant value, it is not relevant for Big Data analysis.

Traditional vs Big Data

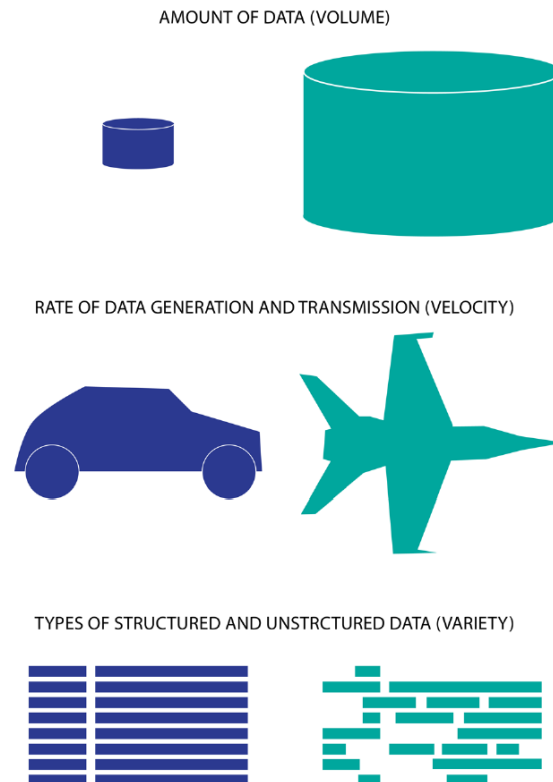


Fig 1.2 Big Data differentiators or 3V's as presented in [24]

One point that is important to emphasize is that Big Data challenges may exist from an individual source, and when aggregated with other sources, the challenge will only increase [26].

In addition to this, Big Data Analytics is concerned with the extraction of value from Big Data, in the form of insights [22]: non-trivial, previously unknown, implicit and potentially useful information.

Big Data adaptation is driven by 3 main factors [24]:

- **Decrease of storage costs:** storage costs has dramatically decreased in the last few years.
- **Big Data tools:** Tools and techniques as Hadoop and NoSQL databases provide the basis technology for an increase in processing speed and complex queries not allowed due to resources limitations.
- **Flexibility:** In traditional data warehouses, users defined schemas ahead of time. With Big Data, users do not have to use predefined formats.

1.7.2. Types of Analytics for Network Forensics

It is also mentioned by [24], and also shown in Figure 1.3, that we can find it in two kinds:

- Batch Processing for data at rest: Also coined "Catch-it-as-you-can" systems by [8]. This approach requires a large amount of storage.
- Stream Processing for data in motion: Also coined "Stop, look and listen" systems by [8]. This approach may require less storage, but it requires a faster processor to keep up with the incoming traffic.

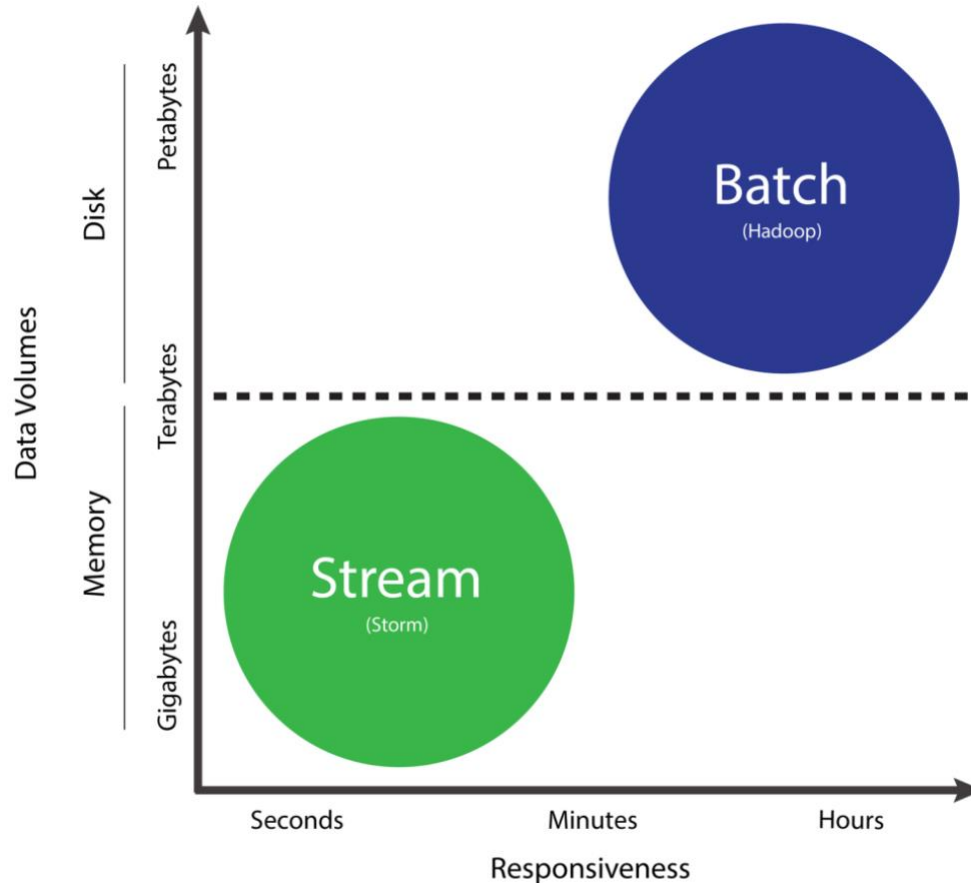


Fig 1.3 Batch processing vs. Stream processing [24]

1.7.3. Big Data Tools and Techniques in Network Forensics for IDS

Considering [14], the following tools and techniques are presented:

- MapReduce: It is a programming model and implementation for processing and generating large data sets, that can be seen as a high-level abstraction of parallel computing [10]. It has the benefit that programs or scripts written in this functional style are automatically parallelized and executed on large cluster of machines, allowing programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distribution system. To explain it briefly: User specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs; and a reduce function that merges all intermediate values associated with the same intermediate key [3], as shown in Figure 1.4. In 2004, Google indicated MapReduce allowed them to read the web in less than 3 hours, instead of four months [27].

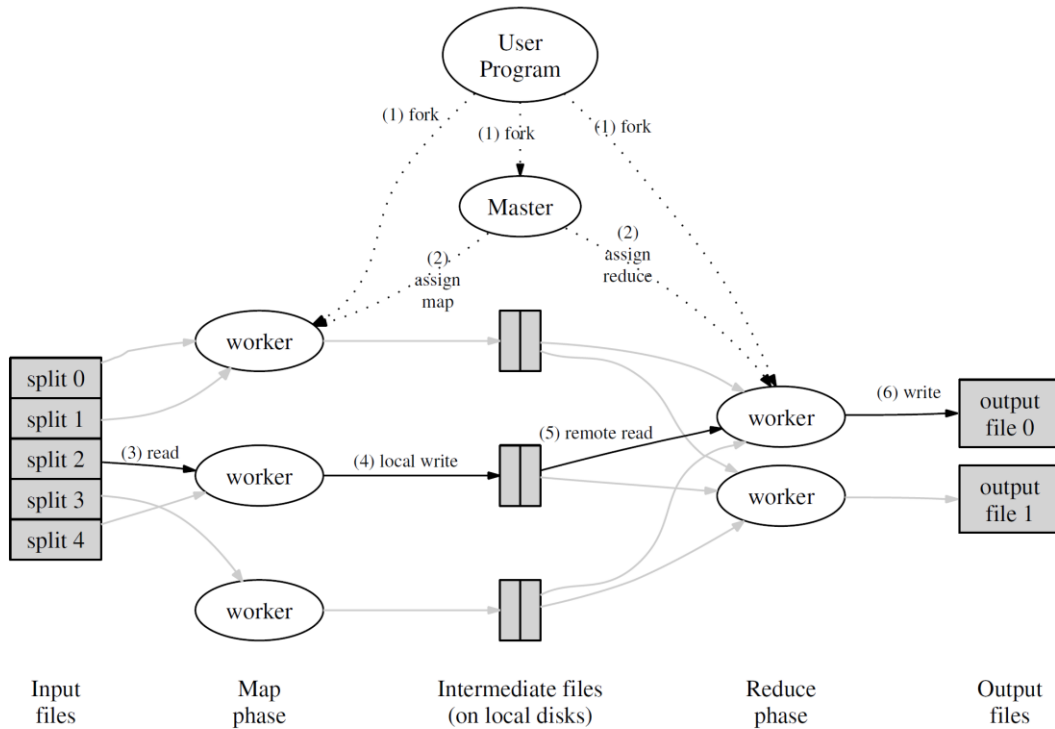


Fig 1.4 Execution overview of MapReduce as discussed in the ground-breaking Google paper [3]

- **Hadoop:** Open source implementation of MapReduce. It is a framework for storing and processing large files, having as its core component the Hadoop Distributed File System (HDFS), where the data is stored. It also has a namenode daemon which maintains the file namespace. The blocks of data are stored on slaves also known as data nodes which guarantee redundancy. In its application side, it has a jobtracker who takes as an input a MapReduce job and which is responsible of task coordination and monitoring the map and reduce tasks [10].
- **Spark:** Distributed data processing solution as Hadoop, but the data processing is stored in memory. Proven sometimes to be 100 times faster than Hadoop [14].
- **Hive:** Open source data warehouse infrastructure running on top of Hadoop. User-friendly interface to Hadoop, and with command in SQL-like form [14].
- **Shark:** Sub-project of Spark, implements Hadoop's Hive on top of Spark. It is fully compatible with Hive [14].
- **Pig:** High level distributed programming on top of Hadoop. Similar to Hive without the SQL query part [14].
- **Storm:** free and open source distributed real time computation system. It processes unbounded streams of data, like a real time Hadoop alike processing [28].

Before entering into the usage examples of the tools and technologies mentioned, two additional related concepts are presented:

- **PageRank Algorithm:** Link analysis algorithm used by Google to weight relative importance of web pages on the internet. Importance of a web

page is determined by: first, how many pages does it contain an hyperlink pointing to it; second, the importance of the pointing pages [10].

- Dependency Graph: Each node represents a host. An edge from point A to B indicates that there exists at least one flow originated from host A towards host B [10].

1.7.4. Usage examples

On [10], an architecture is implemented for Botnet detection based on MapReduce. The algorithm PageRank has to iterate many times, so it is executed over Hadoop. An adjacency matrix of the dependency graph is computed and distributed among all the data nodes before executing the MapReduce. The results are close to 100% true positives and 3% false positives. The limitation in this work is the use of synthetic botnet records.

On [14], a comparison of many Big Data technologies is mentioned: Hadoop, Hive, Pig, Spark and Shark. There are four scenarios discussed, which are data processing tasks among network records to provide a simulation comparative regarding the usage of these scenarios for future cases.

- Scenario 1: Find packets that match a given source IP address and a given source port.
- Scenario 2: Find packets containing a given substring in their payload.
- Scenario 3: Count the number of destination IP per source IP and order the result.
- Scenario 4: Join two sets according to a common key or field, like the source IP address.

The top performers among the four scenarios are Spark and Shark. This study is relevant to give us an awareness of the technologies and their related performances, but the relevance is limited to the scope of the simulation scenarios.

On [29], the author uses Big Data Behavioral analytics in conjunction with graph theoretical concepts with the goal of identifying botnet niches. In their evaluation, they used 100GB of real botnet traffic (Carna botnet as on [30]). Their approach objective can be seen as three steps:

- Identify bots traffic
- Infer botnets from bots traffic
- Pinpoint botnet niches

Regarding their Big Data processing, the authors used two approaches:

- Hadoop
 - Standard approach in the documentation
 - Java implementation
 - MapReduce embedded in this approach
- Multi-Threaded Approach
 - Developed by the authors using C programming language.
 - Packages used: libpcap and libtrace.

Among the comparisons and results, the following points are considered:

- The Multi-Threaded approach processed 1.5 times more throughput than the Hadoop approach.

- The Hadoop approach performed 3 times faster than the Multi-Threaded approach

Regarding the results, the comparison was made between 82 real malicious IP addresses against 10 benign synthetic IP addresses, which is a ratio of traffic not favorable for the benign addresses; this statement is reflected in the results presented:

- 0% False Negative rate for malicious addresses
- 5% False Positive rate for malicious addresses
- 50% False Positive rate for benign addresses. This score wasn't presented in the paper.

CHAPTER 2. STRATOSPHERE IPS

2.1. Motivation

Among the many solutions and literature discussed, Stratosphere IPS is a novel solution based on machine learning algorithms to detect behaviors in a network traffic.

In addition to this, they use real datasets from verified malware and normal traffic connections, which adds relevance to the Network Forensics field and IDS techniques.

2.2. Description of the Stratosphere Project

Stratosphere IPS is a project of research, development and verification methods to detect malware traffic in networks [17]. It was born in from a collaboration of Ph.D. thesis [12]. The Stratosphere Lab is based in the department of Computer Science of the Faculty of Engineering, of the CTU University in Prague.

In the thesis [12], Stratosphere's initial work, the author explains that one of the motivations for the work presented was a better understanding of the behavior of the C&C channel, parting from the hypothesis that flows of a C&C channel are periodic, to support a synchronize effort in the case of an attack.

2.3. Description of SLIPS

Stratosphere Linux Intrusion Prevention System (SLIPS) is the Linux version of the Stratosphere IPS, developed in Python [1]. The latest version of SLIPS includes Markov Chain models for detection of malware behaviors in the network [17].

SLIPS is mainly based on the use of Time Windows, for the following reasons [18]:

- To limit the huge amount of data to be processed
- Botnets tend to have temporal locality behavior, since most actions remain unchanged for several minutes, from 1 to 30 minutes in the tested datasets of SLIPS.
- This allows providing a result in a timely manner to the Network Administrator.

This time windows are of 5 minutes by default in the source code [1]. At the end of which, every host or IP address is labeled.

In a later implementation, as a result of the work in [17], Sliding Detection Windows (SDW) were added to the source code of SLIPS. By default it uses 10

recent time windows to evaluate the traffic and compare it to a threshold score [1].

2.4. Description of Datasets

Datasets of long-term captures can be obtained through a project related to the same lab going by the name of Malware Capture Facility Project [12]. This project consists of a medium scale setup of virtual machines that allowed to continuously infect more than 30 computers for long periods. In some cases, they captured and analyzed botnet behaviors for months. All of the datasets generated are public.

There are two types of dataset used for SLIPS [31], [12]:

- Malware traffic: This is traffic coming from known infected machines and will be the target for the detection. These are for example, Command and Control connections.
- Normal traffic: This is traffic coming from verified normal computes. Very important to find the real performance. The normal behavior of a user cannot be automatized or generated, if this is the case, it would stop being normal anymore.

In addition to this, the collaborators of SLIPS mention the use of Background traffic, which is unverified traffic and that they state it's a traffic of unverified origin [31]. In [31], it is also stated that is important to saturate the algorithms, to verify its memory and performance and to check if algorithms get confused with the data. But for the purpose of training and testing, we consider this type of traffic irrelevant or counter-productive, since it is unverified and does not provide any ground truth and may lead to an error margin in the final result or in the test results.

2.5. Implementation details of SLIPS

SLIPS receives the flows of traffic from Argus. Argus generates bidirectional netflows for SLIPS to process afterward [17]. It is possible for SLIPS to run in real-time according to the collaborators [17].

Argus is used for the purpose of reading the packets from a pcap file or live network, in order to generate the flows [32]. Also, this allows to reduce the volume of unused data and to keep as much of the useful data as possible [12]. And after having generated flows, Stratosphere can process them.

2.5.1. Description of SLIPS processes

SLIPS utilizes two processes in the program itself [17]:

- The first process, reads the flows from Argus and store them in the Multiprocessing Queue. The Multiprocessing Queue is a python type part

of the multiprocessing package, it allows for multi-producer, multi-consumer FIFO (first-in, first-out) queues [33].

- The second process, reads flows from the Queue, computes them, perform some analysis and builds connections from them. For each added flow, the connections are computed and then compared to the Markov Models. This comparison is based on models thresholds, where the minimum distance is looked for. At the end of the Time Window, the alarms found during this period are printed and saved in memory by this second process; which are outputted again at the end of SLIPS.

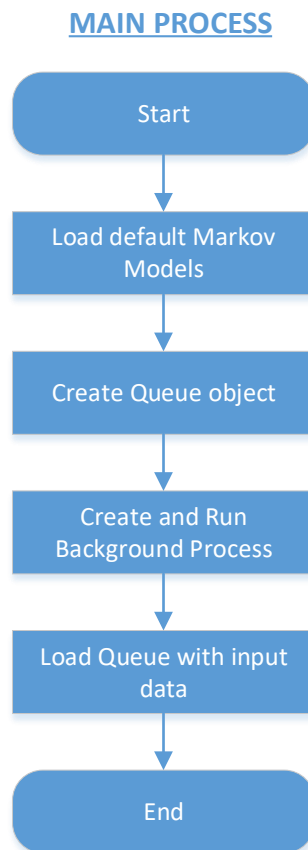


Fig 2.1 Workflow for first process based on the source code of [1]

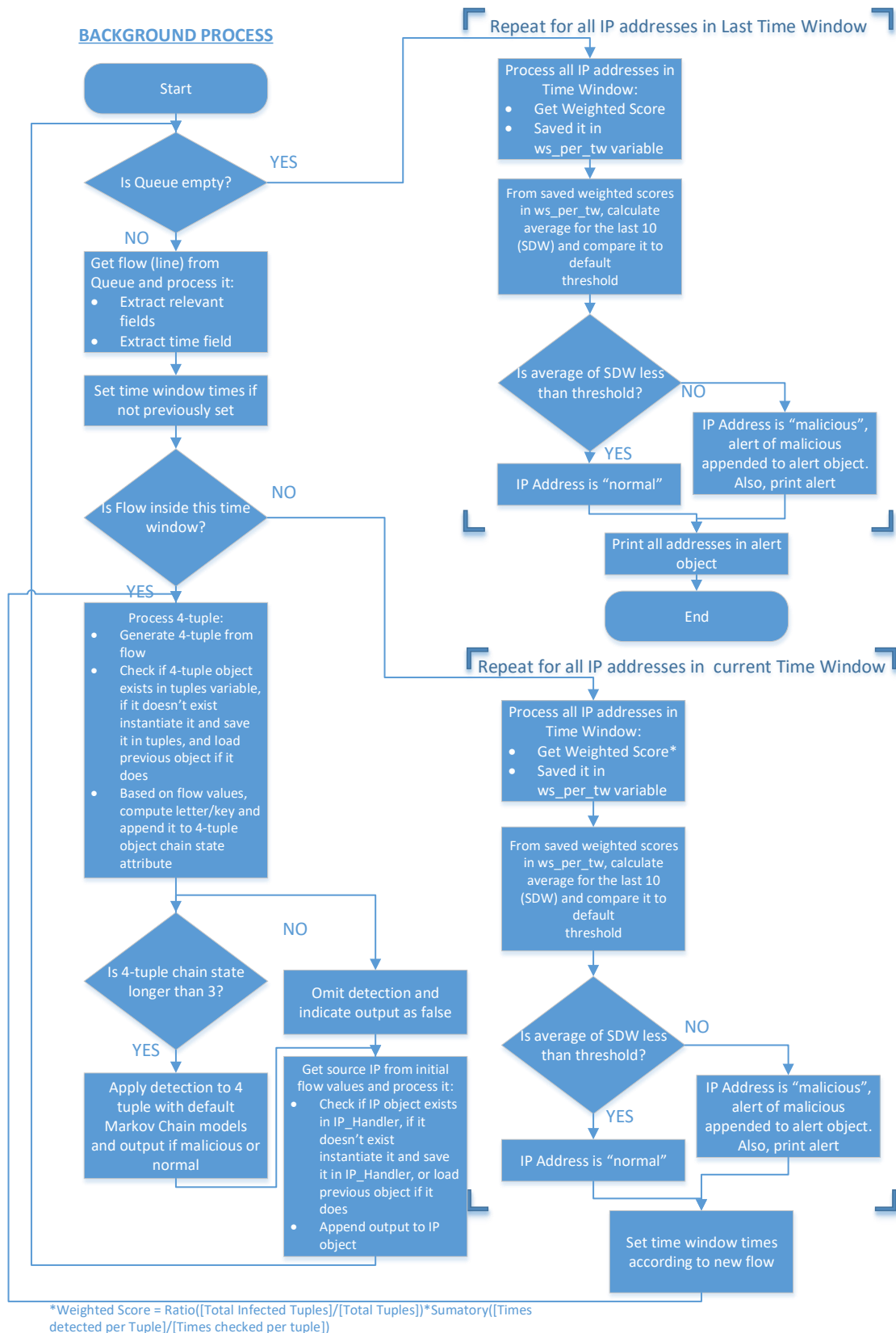


Fig 2.2 Workflow for second process based on source code based on the source code of [1]

2.5.2. Representation of patterns in the network

SLIPS groups the flows according to these criteria:

- The same source IP address.
- The same destination IP address.
- The same destination port.
- The same protocol.

The flows that share these criteria are called 4-tuples or connections. The idea for this type of grouping is to link all connections related to a service; this is why the origin port is omitted [32].

The many flows forming these groups of 4-tuples have a behavior that can be represented using four features are extracted from these flows [32]:

- The size of the flow.
- The duration of the flow.
- The periodicity of the flow.
- The time of the flow.

All the features are self-explanatory, except for the periodicity. The periodicity is calculated in two parts [32] [17].

First, by taking the values of the time stamps from the following flows:

- Current flow, called a.
- Previous flow, called b.
- Two flows ago, called c.

Second, by computing them according to the following three formulas:

$$T1 = a - b \quad (2,1)$$

$$T2 = b - c \quad (2,2)$$

$$TD = |T2 - T1| \quad (2,3)$$

TD is the value used as the periodicity. This value is used since the periodicity has a certain variance that should be accounted for [32].

2.5.3. Computing the patterns as letters

After the four features have been computed as shown in 2.5.2, the states of the flow are generated.

The generation of them is done by the use of some thresholds defined in the following tables; those were defined using the Empirical Cumulative Distribution Function (ECDF) for each of the features for the 33% and 66% of its distribution function [12]:

Table 2.1 Thresholds for labeling the flow based on its size [17]

Size S (Bytes)	< 250	< 1100	>= 1100
Label	Small	Medium	Large

Table 2.2 Thresholds for labeling the flow based on its duration [17]

Duration (s)	< 0.1	< 10	>= 10
Label	Short	Medium	Long

Table 2.3 Threshold for labeling the flow based on its periodicity [17]

Time difference (s)	< 1.05	< 1.3	< 5	>= 5
Periodicity	Strong Periodicity	Weak Periodicity	Weak Non-Periodicity	Strong Non-periodicity

According to the threshold values mentioned previously, a key coding is applied as in the following figure [32].

In addition to this, it is important to mention an additional feature, which is the Symbol for time difference, representing the size of separation in time of the flows, since a time difference of 1 second is not the same as a periodicity of 1 day [32].

	Size Small			Size Medium			Size Large		
	Dur. Short	Dur. Med.	Dur. Long	Dur. Short	Dur. Med.	Dur. Long	Dur. Short	Dur. Med.	Dur. Long
Strong Periodicity	a	b	c	d	e	f	g	h	i
Weak Periodicity	A	B	C	D	E	F	G	H	I
Weak Non-Periodicity	r	s	t	u	v	w	x	y	z
Strong Non-Periodicity	R	S	T	U	V	W	X	Y	Z
No Data	1	2	3	4	5	6	7	8	9

Symbols for time difference:

Between 0 and 5 seconds: .
Between 5 and 60 seconds: ,
Between 60 secs and 5 mins: +
Between 5 mins and 1 hour: *
Timeout of 1 hour 0

Fig 2.3 Key assignment logic in Stratosphere IPS [32]

An example of the flow generated would be the following, related to the 4-tuple 192.168.0.150-46.105.227.94-80-tcp in the capture CTU-Malware-Capture-Botnet-116-4 from the Stratosphere Dataset [32]:

88+H+y+H+H+h+H+H+H+y+y+y+y+y+H+H+y+y+H+y+h+y+y+H+y+y+y+H+h+y+h+H+H+y+h+y+H+y+H+H+y+y+y+H+l+h+y+y+y+y+h+y+y+y+H+H+y+H+y+y+y+y+H+H+H+y+y+y+y+y+y+y+y+h+h+y+h+y+y+y+h+H+H+H+H+H+H+y+H+y+h+y+h+y+h+H+y+y+H+

From the letters and symbols, we can interpret the following [32]:

- Weak and strong periodicity types, and sometimes lost (letter key 'y').
- Time difference of 1 to 5 minutes, related to the periodicity
- Size large present across the chain

2.5.4. Description of the Machine Learning: Markov Chain

Stratosphere later interprets these letters in a model based on a first order Markov Chain. This is a model of the transition probabilities from one state letter to the next one.

If for example we have the following chain of letters and symbols: a,a,c+d+d+
The Markov Chain and matrix would be the following:

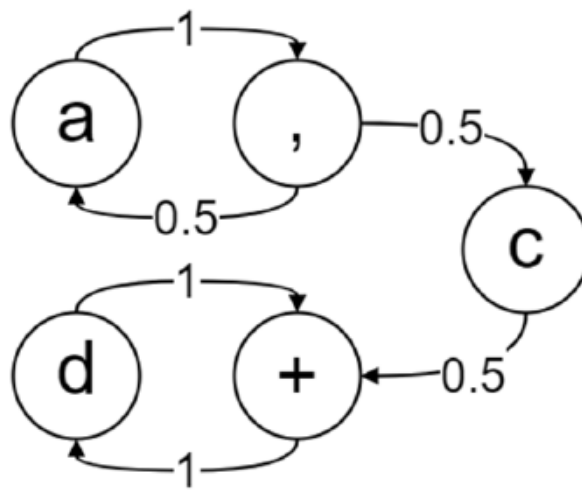


Fig 2.4 Example of Markov Chain for “a,a,c+d+d+” [32]

	a	,	c	+	d
a		1			
,	0.5		0.5		
c				1	
+					1
d				1	

Fig 2.5 Example of Markov Matrix for “a,a,c+d+d+” [32]

The behavioral model for each connection or 4-tuples would be the following [32] [17]:

- Markov Matrix

- Initialization Vector
- The probability of detecting itself
- Threshold of model

The last item of the model, the threshold. It is used as an upper limit for the maximum distance between the model trained and the model evaluated [17] [1]. It is a value that let us know how similar the chains must be to be considered the same behavior, and if not used there will always be some model selected as the best one [12].

2.6. Limitations of current version of SLIPS

Some limitations were addressed for SLIPS in [17]. In the following sections, we will address some additional ones that we consider would help advance the work of SLIPS for the Network Forensics community:

2.6.1. Degradation of performance

There is a substantial degradation of performance on SLIPS.

From a test done with a capture from their dataset named CTU-Malware-Capture-Botnet-168-1 [34], which corresponds to a capture of 3.8 days and 16 different IP addresses involved and a total of 81920 records. To process this capture it took a total of 19298 seconds or 5 hours and 21 minutes in an Ubuntu machine (Ubuntu16.04, 64-bit, Base memory of 3076MB, 2 processors and 40GB of storage).

We present the number of records processed (an approximation to the multiple of 10 for presentation reasons) and the time passed since SLIPS started:

Table 2.4 Amount of records processed vs. Time Passed for dataset CTU-Malware-Capture-Botnet-168-1 in SLIPS

Amount of Records	Time Passed
2290	1 minute
6820	5 minutes
9800	7 minutes
13600	15 minutes
16400	20 minutes
26600	45 minutes
28100	60 minutes
31070	90 minutes
39340	120 minutes
52370	150 minutes
58250	180 minutes
66170	220 minutes

To reinforce the idea from this section: If maintaining the same speed of the first minute, it should have taken 36 minutes; or if maintaining the same speed of the first 15 minutes, it should have taken 2 hours. As more time passes, this degradation of performance becomes more relevant.

If analyzing further, in [33] there is a benchmarking available for the queue class working as part of the multiprocessing package, which used in SLIPS. It's capacity in the same machine used in the previous tests is the following: 83246 average requests per second. This shows us there is an unused capacity available in the machine running SLIPS.

2.6.2. Time windows usage in source code

From analyzing the source code [1], the script part in the processing of 4-tuples before computing the Markov Chain uses the complete 4-tuple accumulated chain. This is the complete concatenation of letters and symbols since the start of the SLIPS program, even before that start of the SDW. This means 4-tuples outside of the time window were being processed. This affects the output in 2 ways:

- All alarms appeared during the first few seconds or minutes of running SLIPS, since there is penalty for computing the distance to the trained Markov Chain models, only the initial records processed have an opportunity of being correctly detected. For example, for the test done in the section 2.6.1, the only alarms appearing were during the first 5 seconds of running SLIPS which correspond to the first 500 records being processed. These alarms were the only ones reported in the final output.
- Since all connections are being processed, the more time it passes, the more records will be gathered for processing, and the longer it will take. This appears to be related to the limitation reported in 2.6.1.

Another important point to mention also related to the source code [1], it is that the time windows are not fixed. They are related to the flow times of the flows present. This also means that all time windows processed will always have traffic. For example, if you have traffic collected of 1 hour, but all of it is allocated in the first 5 minutes and the last 5 minutes, SLIPS will consider only 2 time windows instead of 12. And also, traffic relations are used to compute the chains (section 2.5.2), this means that traffic must be pre-processed before being partitioned for solutions as Spark (section 1.7.4).

2.6.3. Time windows size and convolutional networks

As mentioned in section 2.3, botnets tend to have temporal locality behavior, but this may be shown in 1 minute or 30 minutes [18]. By having only one type of time window, some behaviors are not accounted for or it limits the possibility for their detection.

Taking concepts from Convolutional Neural Networks [35] from the field of Visual Recognition, different layers account for different features; for example, the first layer accounts for a "High Level Perspective", where straight edges, simple colors

and curves are identified. This can be applied in SLIPS, taking into account this first for the trained models themselves, it would be a substantial addition to the design.

2.6.4. Redundancy of periodicity in letters and symbols

In section 2.5.3, it was shown how SLIPS computes patterns as letters. If we analyze this computation, we find that the periodicity is taking into account twice:

- In the periodicity threshold.
- In the symbol for time difference.

This redundancy in the information creates a higher number of characters to be processed, and increase chance of noise to the final result, related to the 5V's in Big Data (section 1.7.1).

It is of value to remark that the periodicity and the time difference are not computed in the same way as shown in section 2.5.2, but their concepts and principles are closely related.

2.6.5. Documentation of thresholds for SDW

An important value used in SLIPS is the weighted score threshold (section 2.5.1), but this is not accounted in any of the main documentation reviewed in this work related to SLIPS: [2], [12], [17], [1].

As mentioned in the first chapter related to the principles of Digital Forensics (section 1.1.1), it is of high important in Digital Forensics to document the processes, in order to allow proper repeatability of the results.

CHAPTER 3. PROPOSAL FOR STRATOSPHERE LINUX IPS

3.1. Motivation

In section 2.6, some limitations to the current version of SLIPS were mentioned. I consider the limitation discussed in section 2.6.1 related to performance the most relevant across all the limitations presented, since with a better performance of SLIPS, the more capability would be available for additional improvements in other aspects, for example those related to section 2.6.3.

In the case of the limitation presented in section 2.6.2, the proposal will account for a response to it and a comparison with the current SLIPS version will be made.

3.2. Description of proposal

The objective of this proposal is to take advantage of the distributing processing available right now in the Python programming language.

Currently in SLIPS, there are only 2 processes, working simultaneously. In this proposal, based on the MapReduce architecture [3], there will be one master process, and many workers or slave processes working simultaneously, these varying on the capacity of processors in the machine.

3.3. Description of package used in proposal

SLIPS is based on Python programming language. Python comes with the availability of the multiprocessing package, which allows any programmer to leverage the multiple processors available on a given machine, being it Windows or Unix [33]. The classes and objects part of this package and used for this proposal are the following:

3.3.1. Proxy Objects

They provide a way to create data which can be shared between different processes. A manager object controls a server process which manages shared objects and return proxies for access. It supports types as lists, dictionaries, between others [33].

This type of objects have lower performance than Queues (mentioned in section 2.6.1), for example:

Table 3.1 Comparison of performance between distinct objects in Ubuntu machine (Ubuntu16.04, 64-bit, Base memory of 3076MB, 2 processors and 40GB of storage) based on the benchmarking script from [33]

Type of Object	Average Requests Per Second
Multiprocessing.Queue Object	83246
Queue managed by Server Process (Proxy Object)	5682
List Object (Python Default)	25896182
List Managed by Server Process (Proxy Object)	14002

From table 3.1, it can be seen that the proxy objects have a lower performance than the other objects, but it allows us with the possibility of having a shared state across distinct processes [33], and without the limitation of the first-in first-out (FIFO) of the Queue.

3.3.2. Pool

This class allows means of parallelizing the execution of a function across processes. In our case, this class represents a pool of worker processes [33].

3.3.3. Cpu_count

An additional function, which is relevant for this work and has been useful for this proposal is to verify the amount of processors through the function `cpu_count()`, which is part of the Multiprocessing package [33].

A common practice is to use two times the amount of processors for the amount of workers, which is used in this work and which has demonstrate to be the right amount during the troubleshooting tests [36].

3.4. Implementation of Proposal

As mentioned in 2.5.1, the main SLIPS source works based on two processes, the main one and the background, and as seen in Fig 2.1 and Fig 2.2, the complexity of the code lies in the background process.

In this proposal, as we are going to see in the following sub-sections, the complexity of the Main Process has been increased in favor of a distribution of data across workers, this is all done based on the current source code of SLIP [1]. The source code of the proposed solution based on MapReduce can be found in Annex A.

3.4.1. Main Process

The main process is in charge of the main following tasks:

- Pre-process all flow records: Flow records are allocated according to Time Windows. This is passed to a List Proxy object, so this data can be shared by many processes.
- Map detection to workers: processed flows are passed to a pool of workers, where each worker or process takes the data of each Time Window separately. The main process waits for the Map detection of all workers to finish processing the complete data.
- Reduce detection output: The output of the Map detection is reduced to an alarm object.
- Print Final alarms: All alarms are printed from the alarms object.

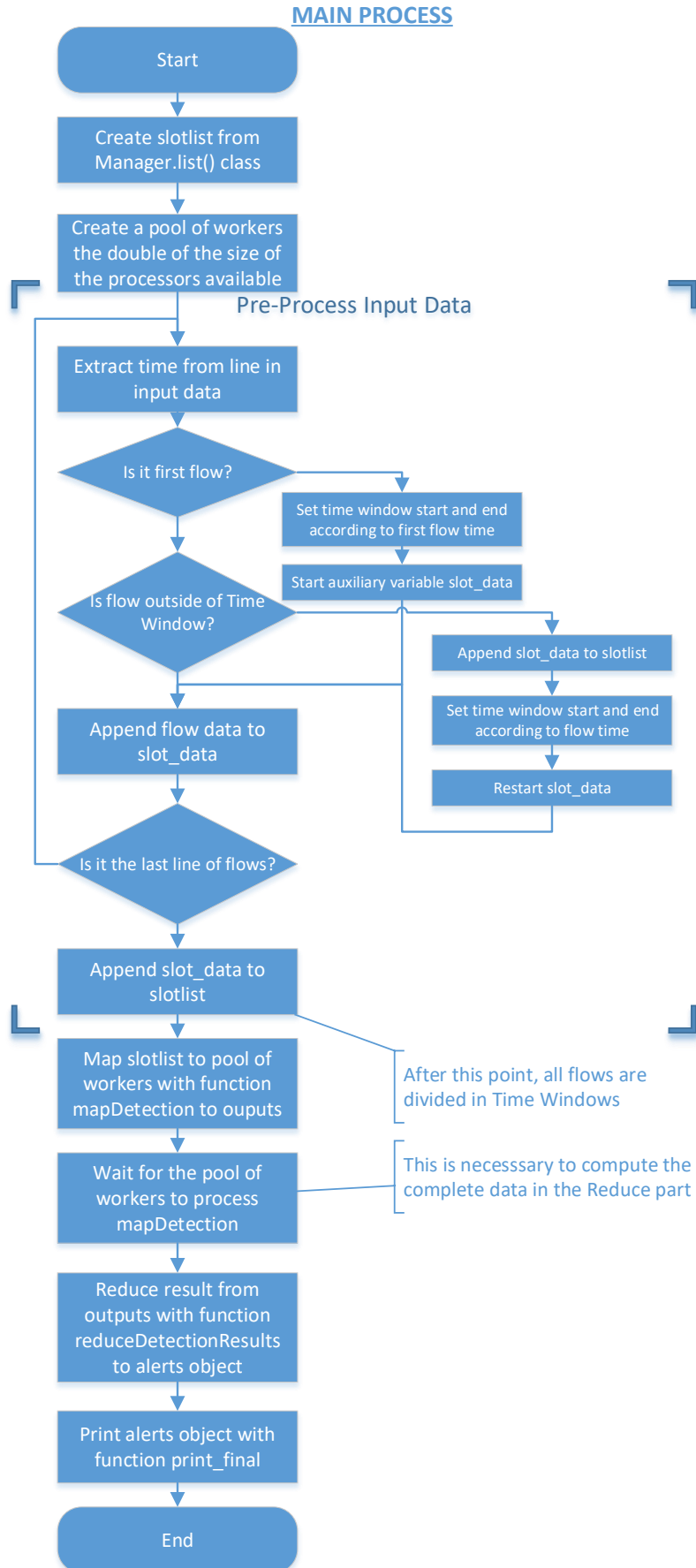


Fig 3.1 Workflow of main process in proposed solution

A further description of the Map and Reduce related functions will be given in the following sub-sections. The final alarm function will be omitted, since it is not relevant for the purpose of this work.

3.4.2. Function: MapDetection

The detection is done according to the default Markov Chain models passed to the function. The function receives as an input the flows of a complete Time Window.

In the Fig 3.2, the workflow of this function is presented. All the detections are aggregated to a variable called ipHandler, which collects the detections per IP. When finished, the Map detection returns an ipHandler object per Time Window,

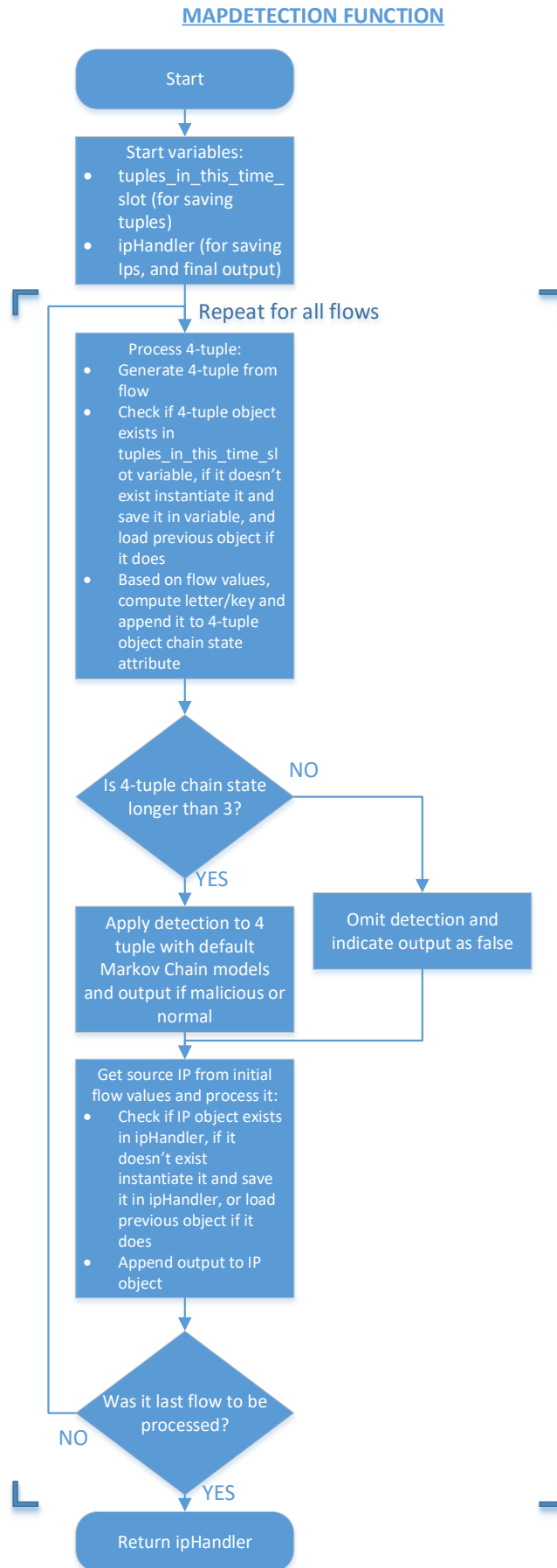


Fig 3.2 Workflow of function MapDetection in proposed solution

3.4.3. Function: ReduceDetectionResults

Having as an input the ipHandler per Time Window, we reduce all this data to one alarm object.

In the Fig 3.3, the workflow of this function is presented. All this data is processed and aggregated to the alarm object, also, the computation for the scores related to the SDW are used in the iterations through each Time Window, in order to determine if any IP address is malicious, to generate, print and aggregate the alarm to the alarms object. The complete list of alarms is saved and returned in the alarms object.

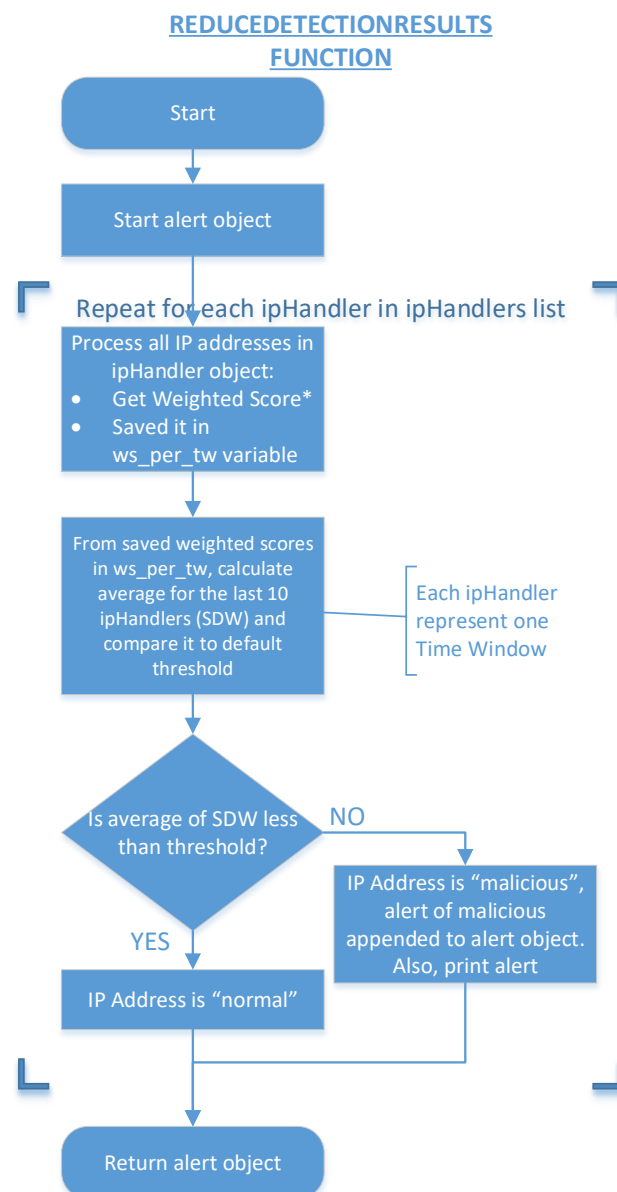


Fig 3.3 Workflow of function ReduceDetectionResults in proposed solution

3.5. Comparison of Tests between Proposal and SLIPS

For the comparison of the performance of the proposal using MapReduce work and SLIPS, the following two public captures will be used [31]:

- CTU-Malware-Capture-Botnet-168-1: network traffic of 3.8 days which includes a machine infected with malware Andromeda, mixed with normal traffic. This capture has a total of 81920 records [34].
- CTU-Malware-Capture-Botnet-169-3: network traffic of 8.13 days which includes a machine infected with malware Miuref, mixed with normal traffic. This capture has a total of 32109 records [37].

This two captures will be run twice, 50% of their records and 100% of them for each solution. This is done to better distinguish performance over time and over the size of the dataset. It is of importance to remark, that both solutions will be using the default models available together with the SLIPS source code [1].

Table 3.2 shows us the amount of Time Windows present across the datasets, taking into account the methodology used by SLIPS as mentioned in section 2.6.2.

The results of these tests are shown in Table 3.3 for SLIPS and in Table 3.4 for the proposed solution using MapReduce. From the data of these tables, two ratio tables are generated; Table 3.5 to compare their performances over time and Table 3.6 to compare the amount of alarms generated for each solution.

Table 3.2 Amount of Time Windows processed for both SLIPS and proposal using MapReduce

Capture	Amount of Time Windows
168-1 at 50%	637
168-1 at 100%	966
169-3 at 50%	581
169-3 at 100%	1937

Table 3.3 Performance of SLIPS with datasets CTU-Malware-Capture-Botnet-168-1 and CTU-Malware-Capture-Botnet-169-3 on Ubuntu machine (Ubuntu16.04, 64-bit, Base memory of 3076MB, 2 processors and 40GB of storage)

Capture	Total IP addresses detected as malicious	Total IP addresses detected	Total alarms generated	Time used in seconds	Time used in minutes
168-1 at 50%	1	14	10	5304.70	88.41
168-1 at 100%	1	16	10	19298.13	321.64

169-3 at 50%	1	85	49	541.93	9.03
169-3 at 100%	1	94	49	934.96	15.58

Table 3.4 Performance of proposal using MapReduce with datasets CTU-Malware-Capture-Botnet-168-1 and CTU-Malware-Capture-Botnet-169-3 on Ubuntu machine (Ubuntu16.04, 64-bit, Base memory of 3076MB, 2 processors and 40GB of storage)

Capture	Total IP addresses detected as malicious	Total IP addresses detected	Total alarms generated	Time used in seconds	Time used in minutes
168-1 at 50%	1	14	445	22.16	0.37
168-1 at 100%	1	16	501	44.60	0.74
169-3 at 50%	1	85	529	22.91	0.38
169-3 at 100%	1	94	577	35.08	0.58

Table 3.5 Ratio between performance times of SLIPS and proposal using MapReduce with datasets CTU-Malware-Capture-Botnet-168-1 and CTU-Malware-Capture-Botnet-169-3 on Ubuntu machine (Ubuntu16.04, 64-bit, Base memory of 3076MB, 2 processors and 40GB of storage)

	Ratio of performance times between SLIPS and Proposal using Map Reduce
168-1 at 50%	239:1
168-1 at 100%	433:1
169-3 at 50%	24:1
169-3 at 100%	27:1

Table 3.6 Ratio between amount of alarms generated by SLIPS and proposal using MapReduce with datasets CTU-Malware-Capture-Botnet-168-1 and CTU-Malware-Capture-Botnet-169-3 on Ubuntu machine (Ubuntu16.04, 64-bit, Base memory of 3076MB, 2 processors and 40GB of storage)

	Ratio of amount of alarms generated between SLIPS and Proposal using Map Reduce
168-1 at 50%	1:45
168-1 at 100%	1:50
169-3 at 50%	1:11
169-3 at 100%	1:12

An important remark regarding these tests is that the IP detected for each of them was the verified infected machine.

From Table 3.5, regarding the performance over time, the proposal using MapReduce performs over time from 24 to 433 times faster than SLIPS, depending on the amount of records evaluated.

From Table 3.6, regarding the alarms generated, SLIPS maintains a constant amount of alarms, for 50% or 100% of the datasets, in contrast to the proposal using MapReduce, which generates an increased amount of alarms when having more traffic as an input. The alarms generated by the proposal are from 11 to 50 times greater than SLIPS, depending on the amount of records evaluated, which indicates there is undetected traffic as reported in the limitation mentioned in section 2.6.2 due to the usage of Time Windows. This additional information points us toward additional insights regarding SLIPS:

- The proposed usage of Time Windows for the detection is a better practice for the SLIPS core detection (section 2.5.4), since it shows an increase amount of alarm for the infected traffic, which indicates an improve detection of the infected traffic.
- For the proposed solution using MapReduce, the alarms are computed regarding the output information from each time window (section 3.4). If comparing the Table 3.2 and the Table 3.4, there are Time Windows with infected traffic that have not been detected, which can be used for improving the default Markov Chain models and adjusting the thresholds of the SLIPS core detection (section 2.5.4).

CONCLUSIONS

The field of Digital Forensics and its branch of Network Forensics have gain relevance for the consequences of their work [9]. Across the different types of attacks inside the Network Forensic field, one that is recurring and becoming wider is the DDoS attack delivered by botnets. The Internet is getting less and less safer, in countries such as China, United States, South Korea or Russia, we find that by the end of 2017 there were more than 450 daily attacks delivered by botnets, with a power of 15.8 million packets per second, and with the demand of ransoms for stopping the attacks in some cases [9].

Related to Network Forensics, we find the IDS solutions, based on signature-based and heuristic approaches. Threats are evolving and IDS together with them. Nowadays, IDS solutions are facing new challenges that weren't there 10 years ago, and which can be tackled with usage of Big Data techniques and tools.

The goal of this work was to demonstrate the usability of Big Data techniques to improve the management of big amounts of data in a real case using real data in the Network Forensics scenario. SLIPS was used in this demonstration, for its use of traffic behaviors, machine learning encompassed with usage of real datasets, since most of the related work is based on synthetic botnet or normal traffic (section 1.7.4 and chapter 2 of [12]).

SLIPS focuses on using two processes, but the work or tasks is not balance or distributed across them, with a degradation of performance over time as seen in 2.6.1. The main process from SLIPS didn't perform any pre-processing for the data, passing only raw traffic data to the internal processing queue.

By using a MapReduce-based approach in this work proposal, to redistribute the SLIPS internal tasks, we can find performance improvements for over 27, up to 433 times in comparison to the original source code and the 100% of records for the datasets presented. The more traffic flows to process, the greater the difference. This was thanks to the redistribution of detections across multiple workers as part of the MapReduce architecture.

An additional benefit of the proposal using MapReduce is the increased amount of alarms, which shows us the continuous infected traffic undetected by SLIPS and an unseen higher efficiency of the SLIPS core detection. As suggested in section 3.5, this additional information can be used to improve the default Markov Chain models and the threshold parameters of SLIPS [1].

The amount of traffic found in the datasets does not justify the use of a Hadoop cluster to distribute the tasks across different machines, the usage of a single server has been enough to demonstrate the leverage of unutilized capacity in

front of us in a single server. Still, for a real life enterprise solution, this would be encouraged, with the benefits of increased capacity and a fault tolerant solution.

An important limitation found for implementing the solution was the usage of Time Windows and chain states, they limit the implementation of the detection to be sequential, this is why for the Reduction part, a wait was implemented. A future option of research would be to perform an additional pre-process of data for the Time Windows, and separate the data on 4-tuples or source IPs, in order to reduce the complexity of data handled by each worker.

As it has been mentioned in section 1.7.4, some Apache Spark implementations are faster than Hadoop MapReduce; an additional example of this, it's a publicly held contest where Spark broke records previously held by Hadoop MapReduce by being 3 times faster with 10 times less resources [38]. A future field of research for SLIPS would be to include the usage of Apache Spark [39] in order to compare the results with the present proposal.

In terms of sustainability considerations, this work will permit the recognition of a better utilization and distribution of resources with the hope of encouraging people to do more with less: a single-server can have dormant capacities which can be enabled with new Big Data approaches.

In terms of ethical considerations: the usage of flow records information did not consider any of the payload, so there is complete respect for the privacy of the information transmitted.

ACRONYMS

C&C	Command and Control
CTU	Czech Technical University
DFRWS	Digital Forensics Research Workshop
DDoS	Distributed Denial of Service
ECDF	Empirical Cumulative Distribution Function
FIFO	First-In, First-Out
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IoT	Internet of Things
IPS	Intrusion Protection System
NoSQL	Non SQL / Not Only SQL
P2P	Peer-to-peer
RFC	Request for Comments
SDW	Sliding Detection Windows
SIEM	Security Information and Event Management
SLIPS	Stratosphere Linux Intrusion Protection System
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

REFERENCES

- [1] S. Garcia, "Stratosphere Linux IPS (slips) Version 0.3.4," GitHub, Inc., [Online]. Available: <https://github.com/stratosphereips/StratosphereLinuxIPS>. [Accessed 17 November 2017].
- [2] "Stratosphere IPS," [Online]. Available: <https://stratosphereips.org/>. [Accessed 19 November 2017].
- [3] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Google, Inc., 2008.
- [4] Collective work of all DFRWS attendees, "A Road Map for Digital Forensic Research," in *The Digital Forensic Research Conference*, Utica, New York, 2001.
- [5] R. Messier, *Network Forensics*, John Wiley & Sons, Inc., 2017.
- [6] G. Alessandro, "Digital Forensics as a Big Data Challenge," *ISSE 2013 Securing Electronic Business Processes*. Springer Vieweg, Wiesbaden, pp. 197-203, 2013.
- [7] N. Kumari and A. K. Mohapatra, "An Insight into Digital Forensics Branches and," in *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, New Delhi, India, 2016.
- [8] "What is network forensics? - Definition from WhatIs.com," [Online]. Available: <http://searchsecurity.techtarget.com/definition/network-forensics>. [Accessed 1 July 2017].
- [9] "DDoS attacks in Q3 2017," [Online]. Available: <https://securelist.com/ddos-attacks-in-q3-2017/83041/>. [Accessed 11 February 2018].
- [10] J. François, S. Wang, W. Bronzi, R. State and T. Engel, "BotCloud: Detecting Botnets Using MapReduce," *IEEE*, Luxembourg, 2011.
- [11] "What is a Botnet?," Kaspersky Lab, [Online]. Available: <https://www.kaspersky.com/resource-center/threats/botnet-attacks>. [Accessed 10 December 2017].
- [12] S. García, "Identifying, Modeling and Detecting Botnet Behaviors in the Network," *Universidad Nacional del Centro de la Provincia de Buenos Aires*, 2014.
- [13] "The Spamhaus Project - The Top 10 Worst Botnet Countries," [Online]. Available: <https://www.spamhaus.org/statistics/botnet-cc/>. [Accessed 7 February 2018].
- [14] S. Marchal, X. Jiang, R. State and T. Engel, "A Big Data Architecture for Large Scale Security Monitoring," in *2014 IEEE International Congress on Big Data*, 2014.
- [15] H. Debar, "An Introduction to Intrusion-Detection Systems," IBM Research, Zurich Research Laboratory, Rüschlikon, Switzerland, 2000.
- [16] K. Nalavade and B. Meshram, "Comparative Study of IDS and IPS," *BIOINFO Computer Engineering*, vol. 1, no. 1, pp. 01-04, 2011.

- [17] O. Lukáš, "Identifying Malicious Hosts by Aggregation of Partial Detections," Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Cybernetics, Prague, 2017.
- [18] S. García, M. Grill, J. Stiborek and A. Zunino, "An Empirical Comparison of Botnet Detection Methods," *Computers & Security*, vol. 45, pp. 100-123, 2014.
- [19] "What is Snort?," Cisco, [Online]. Available: <https://www.snort.org/faq/what-is-snort>. [Accessed 19 November 2017].
- [20] «What is the relationship between Snort and Cisco?,» Cisco Systems, [En línia]. Available: <https://www.snort.org/faq/what-is-the-relationship-between-snort-and-cisco>. [Últim accés: 12 February 2018].
- [21] S. Garcia, «Development of the Stratosphere IPS,» 21 January 2015. [En línia]. Available: <https://stratosphereips.org/development-of-the-stratosphere-ips.html>. [Últim accés: 1 November 2017].
- [22] T. Mahmood and U. Afzal, "Security Analytics: Big Data Analytics for Cybersecurity," in *2013 2nd National Conference on Information Assurance (NCIA)*, 2013.
- [23] R. Jacobson, "2.5 quintillion bytes of data created every day. How does CPG & Retail manage it?," IBM, 24 April 2013. [Online]. Available: <https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/>. [Accessed 12 February 2018].
- [24] Big Data Analytics Working Group, "Big Data Analytics for Security Intelligence," Cloud Security Alliance, 2013.
- [25] "A Case Study In Security Big Data Analysis (2012)," [Online]. Available: <http://www.darkreading.com/monitoring/a-case-study-in-security-big-data-analys/232602339>. [Accessed 12 July 2017].
- [26] R. Zuech, T. Khoshgoftaar and R. Wald, "Intrusion detection and Big Heterogeneous Data: a Survey," *Journal of Big Data*, 2015.
- [27] J. Dean, *Experiences with MapReduce, an Abstraction for Large-Scale Computation*, Google, Inc., 2006.
- [28] "Apache Storm," The Apache Software Foundation, [Online]. Available: <http://storm.apache.org/index.html>. [Accessed 5 February 2018].
- [29] E. Bou-Harb, M. Debbabi and C. Assi, "Big Data Behavioral Analytics Meet Graph Theory: On Effective Botnet Takedowns," IEEE, 2017.
- [30] "Carna Botnet Scans," Center for Applied Internet Data Analysis, [Online]. Available: <https://www.caida.org/research/security/carna/>. [Accessed 15 September 2017].
- [31] "Dataset," [Online]. Available: <https://stratosphereips.org/category/dataset.html>. [Accessed 14 January 2018].
- [32] "Stratosphere IPS. Generation of the Behavioral Models," [Online]. Available: <https://stratosphereips.org/stratosphere-ips-generation-of-the-behavioral-models.html>. [Accessed 17 November 2017].
- [33] "16.6. multiprocessing — Process-based “threading” interface — Python 2.7.14 documentation," [Online]. Available:

- <https://docs.python.org/2/library/multiprocessing.html>. [Accessed 27 January 2018].
- [34] "CTU-Malware-Capture-Botnet-168-1," [Online]. Available: <https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-168-1/>. [Accessed 10 January 2018].
- [35] "A Beginner's Guide To Understanding Convolutional Neural Networks," [Online]. Available: <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/>. [Accessed 20 March 2017].
- [36] D. Hellmann, *The Python Standard Library by Example*, Boston, United States: Addison-Wesley, Pearson Education, Inc., 2011.
- [37] "CTU-Malware-Capture-Botnet-169-3," [Online]. Available: <https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-169-3/>. [Accessed 10 January 2018].
- [38] R. Xin, "Apache Spark the fastest open source engine for sorting a petabyte," 5 November 2014. [Online]. Available: <https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html>. [Accessed 11 February 2018].
- [39] The Apache Software Foundation, «Apache Spark™ - Lightning-Fast Cluster Computing,» The Apache Software Foundation, [En línia]. Available: <https://spark.apache.org/>. [Últim accés: 13 February 2018].
- [40] G. Alessandro, «Digital Forensics as a Big Data Challenge,» *ISSE 2013 Securing Electronic Business Processes*. Springer Vieweg, Wiesbaden, pp. 197-203, 2013.
- [41] Big Data Analytics Working Group, «Big Data Analytics for Security Intelligence,» Cloud Security Alliance, 2013.
- [42] J. François, S. Wang, W. Bronzi, R. State i T. Engel, «BotCloud: Detecting Botnets Using MapReduce,» IEEE, Luxembourg, 2011.
- [43] S. Marchal, X. Jiang, R. State i T. Engel, «A Big Data Architecture for Large Scale Security Monitoring,» de *2014 IEEE International Congress on Big Data*, 2014.
- [44] O. Lukáš, «Identifying Malicious Hosts by Aggregation of Partial Detections,» Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Cybernetics, Prague, 2017.
- [45] S. García, M. Grill, J. Stiborek i A. Zunino, «An Empirical Comparison of Botnet Detection Methods,» *Computers & Security*, vol. 45, pp. 100-123, 2014.
- [46] S. García, «Identifying, Modeling and Detecting Botnet Behaviors in the Network,» Universidad Nacional del Centro de la Provincia de Buenos Aires, 2014.
- [47] T. Mahmood i U. Afzal, «Security Analytics: Big Data Analytics for Cybersecurity,» de *2013 2nd National Conference on Information Assurance (NCIA)*, 2013.
- [48] R. Zuech, T. Khoshgoftaar i R. Wald, «Intrusion detection and Big Heterogeneous Data: a Survey,» *Journal of Big Data*, 2015.

- [49] E. Bou-Harb, M. Debbabi i C. Assi, «Big Data Behavioral Analytics Meet Graph Theory: On Effective Botnet Takedowns,» IEEE, 2017.

ANNEX A

The source code of the proposed solution using MapReduce is the following, with the remark that this is meant to replace only the slips.py, part of the SLIPS source code:

```
#!/usr/bin/python -u
# This file is part of the Stratosphere Linux IPS
# See the file 'LICENSE' for copying permission.
# Author: Sebastian Garcia. eldraco@gmail.com ,
# sebastian.garcia@agents.fel.cvut.cz

#Reference
# 0:starttime, 1:dur, 2:proto, 3:saddr, 4:sport, 5:dir, 6:daddr, 7:dport,
# 8:state, 9:stos, 10:dtos, 11:pkts, 12:bytes

import sys
import os #added by Alfredo
import operator
import signal
from colors import *
from datetime import datetime
from datetime import timedelta
import argparse
import multiprocessing
from multiprocessing import Pool, Manager
import time
from modules.markov_models_1 import __markov_models__
from os import listdir
from os.path import isfile, join
from ip_handler import IpHandler
from utils import SignalHandler
import random
from alerts import *

initial_time = time.time() # starting time
line_number = 0

#check if the log directory exists, if not, create it
logdir_path = "./logs"
if not os.path.exists(logdir_path):
    os.makedirs(logdir_path)
#file for logging
#lognamefile = logdir_path+"/"+ 'log_' + datetime.now().strftime('%Y-%m-%d
%H:%M:%S')+'.txt';
logfile = logdir_path+"/"+ 'logfile' + '.txt';

version = '0.3.5_beta101_alfredo'

def timing(f):
    """ Function to measure the time another function takes."""
    def wrap(*args, **kwargs):
        time1 = time.time()
        ret = f(*args, **kwargs)
        time2 = time.time()
        print '%s function took %0.3f ms' % (f.func_name, (time2-
time1)*1000.0)
        return ret
    return wrap

#Tuple
class Tuple(object):
```

```

""" The class to simply handle tuples """
def __init__(self, tuple4):
    self.id = tuple4
    self.amount_of_flows = 0
    self.src_ip = tuple4.split('-')[0]
    self.dst_ip = tuple4.split('-')[1]
    self.protocol = tuple4.split('-')[3]
    self.state_so_far = ""
    self.winner_model_id = False
    self.winner_model_distance = float('inf')
    self.proto = ""
    self.datetime = ""
    self.T1 = False
    self.T2 = False
    self.TD = False
    self.current_size = -1
    self.current_duration = -1
    self.previous_size = -1
    self.previous_duration = -1
    self.previous_time = -1
    # Thresholds
    self.tto = timedelta(seconds=3600)
    self.tt1 = float(1.05)
    self.tt2 = float(1.3)
    self.tt3 = float(5)
    self.td1 = float(0.1)
    self.td2 = float(10)
    self.ts1 = float(250)
    self.ts2 = float(1100)
    # The state
    self.state = ""
    # Final values for getting the state
    self.duration = -1
    self.size = -1
    self.periodic = -1
    self.color = str
    # By default print all tuples. Depends on the arg
    self.should_be_printed = True
    self.desc = ''
    # After a tuple is detected, min_state_len holds the lower letter
position in the state
    # where the detection happened.
    self.min_state_len = 0
    # After a tuple is detected, max_state_len holds the max letter
position in the state
    # where the detection happened. The new arriving letters to be
detected are between max_state_len and the real end of the state
    self.max_state_len = 0
    self.detected_label = False

    def set_detected_label(self, label):
        self.detected_label = label

    def unset_detected_label(self):
        self.detected_label = False

    def get_detected_label(self):
        return self.detected_label

    def get_state_detected_last(self):
        if self.max_state_len == 0:
            # First time before any detection
            return self.state[self.min_state_len:]
        # After the first detection
        return self.state[self.min_state_len:self.max_state_len]

    def set_min_state_len(self, state_len):
        self.min_state_len = state_len

```

```

def get_min_state_len(self):
    return self.min_state_len

def set_max_state_len(self, state_len):
    self.max_state_len = state_len

def get_max_state_len(self):
    return self.max_state_len

def get_protocol(self):
    return self.protocol

def get_state(self):
    return self.state

def set_verbose(self, verbose):
    self.verbose = verbose

def set_debug(self, debug):
    self.debug = debug

def add_new_flow(self, column_values):
    """ Add new stuff about the flow in this tuple """
    # 0:starttime, 1:dur, 2:proto, 3:saddr, 4:sport, 5:dir, 6:daddr:
    # 7:dport, 8:state, 9:stos, 10:dtos, 11:pkts, 12:bytes
    # Store previous
    self.previous_size = self.current_size
    self.previous_duration = self.current_duration
    self.previous_time = self.datetime
    if self.debug > 2:
        print 'Adding flow {}'.format(column_values)
    # Get the starttime
    self.datetime = datetime.strptime(column_values[0], '%Y/%m/%d
%H:%M:%S.%f')
    # Get the size
    try:
        self.current_size = float(column_values[12])
    except ValueError:
        # It can happen that we dont have this value in the binetflow
        self.current_size = 0.0
    # Get the duration
    try:
        self.current_duration = float(column_values[1])
    except ValueError:
        # It can happen that we dont have this value in the binetflow
        self.current_duration = 0.0
    # Get the protocol
    self.proto = str(column_values[2])
    # increase by 1 amount of flows
    self.amount_of_flows += 1
    # Update value of T1
    self.T1 = self.T2
    try:
        # Update value of T2
        self.T2 = self.datetime - self.previous_time
        # Are flows sorted?
        if self.T2.total_seconds() < 0:
            # Flows are not sorted
            if self.debug > 2:
                print '@',
            # What is going on here when the flows are not ordered?? Are
            # we losing flows?
        except TypeError:
            self.T2 = False
    # Compute the rest
    self.compute_periodicity()
    self.compute_duration()

```

```

        self.compute_size()
        self.compute_state()
        self.compute_symbols()
        if self.debug > 4:
            print '\tTuple {}'.format(self.get_id(), self.amount_of_flows)

    def compute_periodicity(self):
        # If either T1 or T2 are False
        if (isinstance(self.T1, bool) and self.T1 == False) or
        (isinstance(self.T2, bool) and self.T2 == False):
            #self.periodicity = -1
            # Alfredo: error in SLIPS maintained for comparison reasons
            self.periodic = -1
        elif self.T2 >= self.tto:
            t2_in_hours = self.T2.total_seconds() / self.tto.total_seconds()
            for i in range(int(t2_in_hours)):
                self.state += '0'
        elif self.T1 >= self.tto:
            t1_in_hours = self.T1.total_seconds() / self.tto.total_seconds()
            for i in range(int(t1_in_hours)):
                self.state += '0'
        if not isinstance(self.T1, bool) and not isinstance(self.T2, bool):
            try:
                if self.T2 >= self.T1:
                    self.TD = timedelta(seconds=(self.T2.total_seconds() /
                    self.T1.total_seconds())).total_seconds()
                else:
                    self.TD = timedelta(seconds=(self.T1.total_seconds() /
                    self.T2.total_seconds())).total_seconds()
            except ZeroDivisionError:
                # Alfredo: Strongly periodic
                self.TD = 1
            # Decide the periodic based on TD and the thresholds
            if self.TD <= self.tt1:
                # Strongly periodic
                self.periodic = 1
            elif self.TD < self.tt2:
                # Weakly periodic
                self.periodic = 2
            elif self.TD < self.tt3:
                # Weakly not periodic
                self.periodic = 3
            else:
                self.periodic = 4
        if self.debug > 3:
            print '\tPeriodic: {}'.format(self.periodic)

    def compute_duration(self):
        if self.current_duration <= self.td1:
            self.duration = 1
        elif self.current_duration > self.td1 and self.current_duration <=
self.td2:
            self.duration = 2
        elif self.current_duration > self.td2:
            self.duration = 3
        if self.debug > 3:
            print '\tDuration: {}'.format(self.duration)

    def compute_size(self):
        if self.current_size <= self.ts1:
            self.size = 1
        elif self.current_size > self.ts1 and self.current_size <= self.ts2:
            self.size = 2
        elif self.current_size > self.ts2:
            self.size = 3
        if self.debug > 3:
            print '\tSize: {}'.format(self.size)

```

```
def compute_state(self):
    if self.periodic == -1:
        if self.size == 1:
            if self.duration == 1:
                self.state += '1'
            elif self.duration == 2:
                self.state += '2'
            elif self.duration == 3:
                self.state += '3'
        elif self.size == 2:
            if self.duration == 1:
                self.state += '4'
            elif self.duration == 2:
                self.state += '5'
            elif self.duration == 3:
                self.state += '6'
        elif self.size == 3:
            if self.duration == 1:
                self.state += '7'
            elif self.duration == 2:
                self.state += '8'
            elif self.duration == 3:
                self.state += '9'
    elif self.periodic == 1:
        if self.size == 1:
            if self.duration == 1:
                self.state += 'a'
            elif self.duration == 2:
                self.state += 'b'
            elif self.duration == 3:
                self.state += 'c'
        elif self.size == 2:
            if self.duration == 1:
                self.state += 'd'
            elif self.duration == 2:
                self.state += 'e'
            elif self.duration == 3:
                self.state += 'f'
        elif self.size == 3:
            if self.duration == 1:
                self.state += 'g'
            elif self.duration == 2:
                self.state += 'h'
            elif self.duration == 3:
                self.state += 'i'
    elif self.periodic == 2:
        if self.size == 1:
            if self.duration == 1:
                self.state += 'A'
            elif self.duration == 2:
                self.state += 'B'
            elif self.duration == 3:
                self.state += 'C'
        elif self.size == 2:
            if self.duration == 1:
                self.state += 'D'
            elif self.duration == 2:
                self.state += 'E'
            elif self.duration == 3:
                self.state += 'F'
        elif self.size == 3:
            if self.duration == 1:
                self.state += 'G'
            elif self.duration == 2:
                self.state += 'H'
            elif self.duration == 3:
                self.state += 'I'
```

```

elif self.periodic == 3:
    if self.size == 1:
        if self.duration == 1:
            self.state += 'r'
        elif self.duration == 2:
            self.state += 's'
        elif self.duration == 3:
            self.state += 't'
    elif self.size == 2:
        if self.duration == 1:
            self.state += 'u'
        elif self.duration == 2:
            self.state += 'v'
        elif self.duration == 3:
            self.state += 'w'
    elif self.size == 3:
        if self.duration == 1:
            self.state += 'x'
        elif self.duration == 2:
            self.state += 'y'
        elif self.duration == 3:
            self.state += 'z'
elif self.periodic == 4:
    if self.size == 1:
        if self.duration == 1:
            self.state += 'R'
        elif self.duration == 2:
            self.state += 'S'
        elif self.duration == 3:
            self.state += 'T'
    elif self.size == 2:
        if self.duration == 1:
            self.state += 'U'
        elif self.duration == 2:
            self.state += 'V'
        elif self.duration == 3:
            self.state += 'W'
    elif self.size == 3:
        if self.duration == 1:
            self.state += 'X'
        elif self.duration == 2:
            self.state += 'Y'
        elif self.duration == 3:
            self.state += 'Z'

def compute_symbols(self):
    if not isinstance(self.T2, bool):
        if self.T2 <= timedelta(seconds=5):
            self.state += '.'
        elif self.T2 <= timedelta(seconds=60):
            self.state += ','
        elif self.T2 <= timedelta(seconds=300):
            self.state += '+'
        elif self.T2 <= timedelta(seconds=3600):
            self.state += '*'
    if self.debug > 3:
        print '\tTD:{}, T2:{}, T1:{}, State: {}'.format(self.TD, self.T2,
self.T1, self.state)

def get_id(self):
    return self.id

def __repr__(self):
    return '{} [{}] ({}): {}'.format(self.color(self.get_id()), self.desc,
self.amount_of_flows, self.state)

def print_tuple_detected(self):
    """

```

```

        Print the tuple. The state is the state since the last detection of
        the tuple. Not everything
        """
        return('{} [{}] ({}): {} Detected as:
        {}'.format(self.color(self.get_id()), self.desc, self.amount_of_flows,
        self.get_state_detected_last(), self.get_detected_label()))

    def set_color(self, color):
        self.color = color

def mapDetection(slotline, verbose = 1, debug = 0, whois = False):
    ipHandler = IpHandler(verbose, debug, whois)
    current_index = slotline['index']
    #print 'the current index is {}'.format(current_index)
    tuples_in_this_time_slot = {}
    values = slotline['values']
    for flow_values in values:
        #print flow
        tuple4 = flow_values[3] + '-' + flow_values[6] + '-' + flow_values[7]
+ '-' + flow_values[2]
        flowtime = datetime.strptime(flow_values[0], '%Y/%m/%d %H:%M:%S.%f')
        #Review if tuple already exists
        if tuple4 in tuples_in_this_time_slot:
            tuple = tuples_in_this_time_slot[tuple4]
        else:
            tuple = Tuple(tuple4)
            tuple.set_verbose(verbose)
            tuple.set_debug(debug)
            tuples_in_this_time_slot[tuple4] = tuple
        #Add flow and compute chain states
        tuple.add_new_flow(flow_values)

        ###Let's detect
        (detected, label, statelen) = __markov_models__.detect(tuple, verbose,
debug)
        #print 'detected is {}'.format(detected)
        if detected:
            # Change color
            tuple.set_color(magenta)
            # Set the detection label
            tuple.set_detected_label(label)
        elif not detected:
            tuple.unset_detected_label()
        ###End of detected

        ###Add detection
        tuples_in_this_time_slot[tuple4] = tuple
        ip_address = ipHandler.get_ip(flow_values[3])
        ip_address.add_detection(tuple.detected_label, tuple.id,
tuple.current_size, flowtime, flow_values[6], tuple.get_state_detected_last(),
current_index)
        #return current index
        ###Return IpHandler, to have it processed later
        return ipHandler

def reduceDetectionResults(ipHandlers, slotlist, verbose = 1, debug = 0, whois
= False, sdw_width = 10, threshold = 0.002):
    alerts= {}
    for index,ipHandler in enumerate(ipHandlers):
        for address in ipHandler.addresses.values():
            # Change to keys to print addresses
            # print address
            start_time = slotlist[index]['start']
            end_time = slotlist[index]['end']

            #print 'address is {}'.format(address.address)

```

```

    ###Function Get Verdict
    address.get_weighted_score(start_time, end_time, index)
    #Check if any traffic in TW, kind of irrelevant since TWs are not
fixed
    if len(slotlist[index]['values']) > 0:
        startindex = index - sdw_width # compute SDW indices
        if startindex < 0:
            startindex = 0
        sdw = []
        #Looking for the WS in all the previous windows
        for i in range(startindex, index): # fill the sdw
            if address.address in ipHandlers[i].addresses:
                if
ipHandlers[i].addresses[address.address].ws_per_tw.has_key(i):

sdw.append(ipHandlers[i].addresses[address.address].ws_per_tw[i])
        # If it doesn't have the key? Add a try
        ### Alfredo: ERROR, if swd_width is larger than total windows,
lower mean, but maintain for comparison reasons
        mean = sum(sdw) / float(sdw_width)
        # Did we detect it?
        if mean < threshold:
            # No
            address.last_verdict = "Normal"
            address.last_SDW_score = mean;
            if address.address not in alerts:
                alerts[address.address] = []

        else:
            # Yes
            address.alerts.append(IpDetectionAlert(datetime.now(),
address.address, mean))
            address.last_verdict = "Malicious"
            address.last_SDW_score = mean
            if address.address in alerts:

alerts[address.address].append(IpDetectionAlert(datetime.now(),
address.address, mean))
            else:
                alerts[address.address] = []

alerts[address.address].append(IpDetectionAlert(datetime.now(),
address.address, mean))
            else:
                # self.last_verdict = None
                address.last_verdict = 'Unknown'
                print 'we got an unknown!'

        #####End of Verdict

        if address.last_verdict.lower() == 'malicious' and verbose > 0:
            print red("\t+{} verdict: {} (SDW score: {:.5f}) | TW weighted
score: {} = {} x {}".format(address.address, address.last_verdict,
address.last_SDW_score, address.last_tw_result[0], address.last_tw_result[1], add
ress.last_tw_result[2]))

        return alerts

def print_final(alerts):
    detected_counter = 0
    print '\nFinal Alerts generated:'
    f = open(logfile, "w")
    f.write("DATE:\t{}\nSummary of addresses in this
capture:\n\n".format(datetime.now().strftime('%Y/%m/%d %H:%M:%S')))
    f.write('Alerts:\n')
    for ip, alertsforIp in alerts.items():
        if len(alertsforIp) > 0:
            detected_counter+=1

```

```

        print "\t - " + ip
        f.write( '\t - ' + ip + '\n')
    for alert in alertsforIp:
        print "\t\t" + str(alert)
        f.write( '\t\t' + str(alert) + '\n')

    s = "{} IP(s) out of {} detected as
malicious.".format(detected_counter,len(alerts))
    f.write(s)
    print s
    f.close()

#####
# Main
#####
if __name__ == '__main__':
    print 'Stratosphere Linux IPS. Version {}'.format(version)
    print('https://stratosphereips.org')
    print

    # Parse the parameters
    parser = argparse.ArgumentParser()
    parser.add_argument('-a', '--amount', help='Minimum amount of flows that
should be in a tuple to be printed.', action='store', required=False,
type=int, default=-1)
    parser.add_argument('-v', '--verbose', help='Amount of verbosity. This
shows more info about the results.', action='store', default=1,
required=False, type=int)
    parser.add_argument('-e', '--debug', help='Amount of debugging. This shows
inner information about the flows.', action='store', default=0,
required=False, type=int)
    parser.add_argument('-w', '--width', help='Width of the time slot used for
the analysis. In minutes.', action='store', default=5, required=False,
type=int)
    parser.add_argument('-d', '--datawhois', help='Get and show the WHOIS info
for the destination IP in each tuple', action='store_true', default=False,
required=False)
    parser.add_argument('-D', '--dontdetect', help='Dont detect the malicious
behavior in the flows using the models. Just print the connections.',
default=False, action='store_true', required=False)
    parser.add_argument('-f', '--folder', help='Folder with models to apply
for detection.', action='store', required=False)
    parser.add_argument('-s', '--sound', help='Play a small sound when a
periodic connections is found.', action='store_true', default=False,
required=False)
    parser.add_argument('-t', '--threshold', help='Threshold for detection
with IPHandler', action='store', default=0.002, required=False, type=float)
    parser.add_argument('-S', '--sdw_width', help='Width of sliding window.
The unit is in \time windows\'. So a -S 10 and a -w 5, means a sliding window
of 50 minutes.', action='store', default=10, required=False, type=int)
    parser.add_argument('-W', '--whitelist', help="File with the IP addresses to
whitelist. One per line.", action='store', required=False)

    args = parser.parse_args()

    # Check the verbose level
    if args.verbose < 1:
        args.verbose = 1

    # Check the debug level
    if args.debug < 0:
        args.debug = 0

    if args.dontdetect:
        print 'Warning: No detections will be done. Only the behaviors are
printed.'
        print

```

```

# If the folder with models was specified, just ignore it
args.folder = False

# Do we need sound?
if args.sound:
    import pygame.mixer
    pygame.mixer.init(44100)
    pygame.mixer.music.load('periodic.ogg')

# Read the folder with models if specified

# added by Alfredo 070218
#args.folder = "C:/Users/Alfredo/Google
Drive/MASTEAM/Thesis/workspace/spark_v1/models"
#args.folder = "/media/sf_shared_vm/workspace/spark_v2/"
###
if args.folder:
    #print 'got here'
    onlyfiles = [f for f in listdir(args.folder) if
isfile(join(args.folder, f))]
    if args.verbose > 2:
        print 'Detecting malicious behaviors with the following models:'
    for file in onlyfiles:
        __markov_models__.set_verbose(args.verbose)
        __markov_models__.set_debug(args.debug)
        __markov_models__.set_model_to_detect(join(args.folder, file))
        #added by Alfredo 070218
        #print 'alfredo comment: markov model added'

# Read whitelist
whitelist = set()
if args.whitelist:
    try:
        #whitelist = set()
        content = set(line.rstrip('\n') for line in open(args.whitelist))
        if len(content) > 0:
            if args.verbose > 1:
                #if True:
                print blue("Whitelisted IPs:")
                for item in content:
                    if args.verbose > 1:
                        #if True:
                        print blue("\t" + item)
            whitelist = content
        except Exception as e:
            print blue("Whitelist file '{}' not
found!".format(args.whitelist))

###Ubuntu VirtualBox Datasets
filename = "/media/sf_shared_vm/workfiles/169_3_2016-08-03_win4.binetflow"
#filename = "/media/sf_shared_vm/workfiles/169_3_2016-08-
03_win4_half.binetflow"

#filename = "/media/sf_shared_vm/workfiles/168_1_2016-07-30_capture-
win1.binetflow"
#filename = "/media/sf_shared_vm/workfiles/168_2_2016-08-03_win-
1.binetflow"
#filename = "/media/sf_shared_vm/workfiles/168_1_2016-07-30_capture-
win1_half.binetflow"

###Alpha2: Manager setup
manager = Manager()
slotlist = manager.list()
pool_size = multiprocessing.cpu_count() * 2

```

```
#####

###Alpha2: Data preprocess
### We are adding each column an stripping it from unnecessary content
with open(filename) as file:
    slot_data = {}
    timewindow_index = 0
    count_lines = 0
    slot_start = -1
    slot_end = -1
    slot_width_minutes = timedelta(minutes=5)
    for line in file:
        #try:++
        #print line
        #print count_lines
        column_values = line.strip('[').split(',')[13]
        #print column_values
        try:
            flow_start = datetime.strptime(column_values[0], '%Y/%m/%d
%H:%M:%S.%f')
            if slot_start == -1:
                slot_start = flow_start
                slot_end = slot_start + slot_width_minutes
                slot_data['start'] = slot_start
                #print slot_data['start']
                slot_data['end'] = slot_end
                slot_data['values'] = []
                slot_data['index'] = timewindow_index
                #print 'slot_data is {}'.format(slot_data)
            elif flow_start >= slot_end:
                slotlist.append(slot_data)
                slot_start = flow_start
                slot_end = slot_start + slot_width_minutes
                timewindow_index += 1
                #print 'slotlist is {}'.format(str(slotlist))
                slot_data['start'] = slot_start
                slot_data['end'] = slot_end
                slot_data['values'] = []
                slot_data['index'] = timewindow_index
                slot_data['values'].append(column_values)
                #print len(slotlist)
                #print slotlist
                count_lines += 1
            except ValueError:
                #print 'Error in line'
                #queue.put('stop')
                continue
        #last slot to be appended
        slotlist.append(slot_data)
    ###Alpha 2 comment: file uploaded completely

    ###Alpha 2: time to process data

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                maxtasksperchild=1,
                                )
    pool_outputs_ipHandlers = pool.map(mapDetection, slotlist)
    pool.close() # no more tasks
    pool.join() # wrap up current tasks
    #print len(pool_outputs_ipHandlers)
    #print pool_outputs_ipHandlers

    #Execution time before Reduce
    clockTime = time.time() - initial_time # execution time
    timepassed = "Elapsed time for detection only is {}".format(clockTime)
```

```
print timepassed

alerts = reduceDetectionResults(pool_outputs_ipHandlers, slotlist)
print_final(alerts)

#Execution time
clockTime = time.time() - initial_time # execution time
timepassed = "Total elapsed time is {}".format(clockTime)
print timepassed
```