



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Institut für
Technische Informatik und
Kommunikationsnetze

Mahdi Asadpour

Ground truth analysis of anomalies in traffic feature distributions

Research in Computer Science II (263-0600-00L)
September 2009

Tutor: Bernhard Tellenbach
Supervisor: Prof. Bernhard Plattner

Abstract

Recently, Traffic Entropy Spectrum (TES) has been proposed as a promising tool to analyze the changes in traffic distributions, as a basis for detecting anomalies in a network system. TES employs Tsallis entropy that offers a more detailed view on the changes in the underlying distributions than the previous one, Shannon entropy.

Evaluation and approach used for the detection of anomalies in TES are incomplete and the TES method may come with high ratio of false positives or even false negatives, hence need to be investigated in-depth.

The goal of this work is to make an in-depth analysis of TES by taking its results and cross-checking them with the output of another program FlowSketches/FlowExtractor, which uses a different measurement (KL-distance). A program as an extension of FlowSketches is also developed in this work. It takes the anomaly information, analyzes them, and as the result it gives out a good possible reason for every anomaly. Finally, this program inserts all the feature information into a designed MySQL database for future queries.

Contents

1	Introduction	11
1.1	Motivation	11
1.2	Related Work	11
1.2.1	Entropy	12
1.2.2	Traffic Entropy Spectrum (TES)	12
1.3	The Task	12
1.3.1	Data	14
1.4	Overview	14
2	Problem and Approaches	15
2.1	Problem	15
2.2	Other Available Tools: FLAME	15
2.2.1	FlowExtractor	16
2.2.2	FlowSketches	16
2.3	Approach	17
2.3.1	AnalyzeTES program	18
2.3.2	Finding the anomalies	18
2.3.3	Pre-processing	20
2.3.4	Analysis	21
2.3.5	Post-processing	23
2.3.6	Writing the result	23
2.4	Summing up	24
3	Design and Implementation	25
3.1	Database Design	25
3.1.1	Anomaly database table	25
3.1.2	Attack database table	26
3.1.3	NetScanAttack database table	27
3.1.4	AttacksInfo database table	28
3.2	Program design	28
3.2.1	Main function	28
3.2.2	Anomaly class	29
3.2.3	Database class	32
3.2.4	Utility class	32
3.2.5	Base flowchart and summing up	33
3.3	Installing and Usage the program	34
3.3.1	Install	34
3.3.2	Usage	36
4	Experimental Result	37
4.1	First approach: examining an anomaly from TES	37
4.2	Second approach: example of false negative	41
5	Outlook	45
6	Summary and Conclusion	47

A	Implementation details	49
A.1	Main	49
A.1.1	run function	49
A.1.2	autoRunning function	49
A.2	Anomaly class	49
A.2.1	init: first function	52
A.2.2	init: second function	52
A.2.3	isInside function	52
A.2.4	readAnomalies function	52
A.2.5	printAnomalies function	53
A.2.6	printTimes function	53
A.2.7	processAnomalies function	53
A.2.8	postProcessAnomalies function	53
A.2.9	analyze function	53
A.2.10	process function	54
A.2.11	printAttacks function	54
A.2.12	printNetScanAttacks function	54
A.2.13	writeDBScriptUnprocessed function	55
A.2.14	writeDBScriptProcessed function	55
A.2.15	writeDBScriptNetScan function	55
A.2.16	writeDBScript function	55
A.2.17	setTsallisEntropy function	56
A.2.18	checkForNetScanAttack function	56
A.2.19	getNoVictimsByAttacker function	56
A.3	Database class	57
A.3.1	Database constructor	58
A.3.2	Database destructor	58
A.3.3	testDB function	58
A.3.4	startDB: first function	58
A.3.5	startDB: second function	59
A.3.6	startDBTES function	59
A.3.7	getError function	59
A.3.8	getMedianTsallisValue function	59
A.3.9	readDBScriptAnomaly function	60
A.3.10	readDBScriptAttack function	60
A.3.11	readDBScriptNetScanAttack function	60
A.3.12	readDBScriptCreate function	60
A.3.13	finishDB function	60
A.3.14	execDBScriptAnomaly function	61
A.3.15	execDBScriptAttack function	61
A.3.16	execDBScriptNetScanAttack function	61
A.3.17	execDBScriptCreate function	61
A.4	Utility class	62
A.4.1	toString function	62
A.4.2	fromString function	62
A.4.3	groIPs function	62
A.4.4	test function	63
A.4.5	dottedIP function	63
A.4.6	readableTime function	63
B	Abbreviations	65
C	Timetable	67

D Task Description	69
D.1 Introduction	69
D.2 Available data	69
D.2.1 Cluster and NetFlow Data Set	69
D.3 The Task	70
D.3.1 Identify anomalous peaks in a netflow trace	70
D.3.2 Investigation of root cause for (anomalous) peaks	70
D.3.3 Optional: Improvement and tuning of the anomaly detection approach	70
D.4 Deliverables	70
D.4.1 Documentation structure	71
D.4.2 Presentations	71
D.5 General Information	71

List of Figures

1.1	TES visualization tools.	13
2.1	A snapshot of TES visualization: for TCP source IP, the incoming traffic (rectangles are not from TES, added for highlighting).	17
2.2	Snapshot of (a) Sample running of FlowSketches, (b) Its result, (c) Result from intermediate step, (d) Outcome of FlowExtractor.	18
3.1	Database design: relation and fields of tables.	26
3.2	Class dependency.	28
3.3	Call graph of main function.	29
3.4	Dependency graph of main function.	29
3.5	Dependency graph of Anomaly class.	29
3.6	Dependency graph of Database class.	32
3.7	Dependency graph of Utility class.	33
3.8	AnalyzeTES flow diagram.	35
4.1	A snapshot of TES visualization program, for incoming traffic of source IP addresses.	38
4.2	A snapshot of TES visualization program, for incoming traffic of destination IP addresses.	38
4.3	A possible anomaly discovered by TES: (a) for the source IP addresses, (b) for the destination IP addresses.	39
4.4	A snapshot from the output result of FlowExtractor (IP addresses are anonymized).	40
4.5	A UDP Flooding attack occurred in period that the black rectangle indicates, but it is not reflected by TES.	43
A.1	Call graph of Anomaly::process function.	54
A.2	Call graph of Anomaly::setTsallisEntropy function.	56
D.1	Working cluster structure at TIK	72

List of Tables

1.1	Information about the existing NetFlow data set	14
2.1	Considered attacks, part one.	21
2.2	Considered attacks, part two.	22
B.1	List of abbreviations used in text.	65
C.1	Tasks and weeks used for them.	67
D.1	Facts about our NetFlow data set	70

Chapter 1

Introduction

In this chapter, we deal with the motivation of doing such a project, related works in this field, and what tasks should be done as the result.

1.1 Motivation

Network operators (such as ISPs, University networks like SWITCH [6]) may confront with a lot of unusual behaviors: ranging from a huge amount of requests from local network, to different calls from outside networks in a big amount. We name these unusual behaviors in a network as anomalies. These anomalies can span a vast range of events:

- Network abuse, e.g. DOS attacks, IP/Port scans, Worms, etc
- Equipment failures, e.g. Server panic, Router outage, etc
- Unusual user behavior, e.g. Flash crowds, etc
- Unknown events

The operators need to (automatically) detect these anomalies to be able to defend, as soon as possible. But with the huge traffic of data in network, the real time monitoring and analysis of the traffic data become a challenging task. So far, two approaches to network anomaly detection have been proposed:

1. Signature-based approach: that is used for detection based on the anomalies' signature known in advance.
2. Statistics-based approach: that does not require prior knowledge and can work even for new anomalies.

The statistics based approach has attracted considerable attention due to its capability in discovering new anomalies. A very important component of this approach is *change detection*, which can be done at different levels: Traffic volume level (contains number of bytes or packets), and Traffic feature/flow level (usually characterized by source and destination IP addresses, source and destination port numbers, and protocol number). For the purpose of detecting changes, we usually build a model for normal user behavior in learning phase, and any inconsistent behavior with this model is considered as anomaly.

In this work, we focus on the statistics-based approach at traffic flow level (in Cisco NetFlow format [5]).

1.2 Related Work

Efficient and accurate change detection in network systems is a challenging task and has been considered in many research papers.

1.2.1 Entropy

A prominent technique is to use the entropy analysis for network anomaly detection. Entropy analysis not only reduces the amount of stored information, also allows for a compact visualization of changes.

Shannon entropy [17], which has been promisingly employed by several works [12, 15, 20], reduces the information about a distribution to a single number. Doing so, the nature cause of an anomaly may be deleted or a certain group of anomalies may be overlooked [20, 18].

Ziviani et al. [20] used non-extensive entropy technique (Tsallis entropy [19]), that offers a more detailed view on the changes. They also showed that this entropy is better suited to capture the network traffic characteristics of Denial of Service (DoS) attacks. Equation 1.1 is the discrete version of Tsallis entropy, and parameter q is a measure of the non-extensivity of the system of interest.

$$S_q(p) = \frac{1}{q-1} \left(1 - \sum_{k=1}^n p^q(k) \right) \quad (1.1)$$

Different values of q in equation 1.1 highlight various activity regions in the feature elements:

1. $q < 1$: it happens when many elements each with a low activity are captured. This case mostly highlights anomalies with distributed nature on many elements.
2. $q \simeq 1$ ¹: it happens when a large number of elements with similar activities are captured.
3. $q > 1$: it happens when element with high activity are captured. This case mostly highlights anomalies targeting a few servers with a high traffic volume.

1.2.2 Traffic Entropy Spectrum (TES)

Recently, Tellenbach et al. [18] introduced the Traffic Entropy Spectrum (TES) to analyze the changes in Tsallis entropy and to use them as a basis for an anomaly detection system. TES captures and visualizes important characteristics requiring little or no tuning to specific attacks. In order to calculate TES out of a huge amount of Cisco NetFlow data of SWITCH network, the compressed NetFlow data is first decompressed and the data from all sources are merged together. After the merging, a sorter comes into play to sort the incoming flows according to their starting time. TES then calculates the flow, packet and byte count, as well as the Tsallis entropy with different q values for available NetFlow features. Finally, the output is written back to disk for storage and analysis using a database environment. Besides, TES visualization tools (written in MATLAB) [11] provides a graphical representation of the stored data (see Figure 1.1). In TES, the same as Tsallis entropy, various q values may highlight different anomalies. TES though uses different colors and hence patterns to highlight the changes, visually.

1.3 The Task

Evaluation as well as the approach used for the detection of anomalies in TES are incomplete since they only focused on the exposure of a few large scale anomalies. As Tellenbach et al. strictly mentioned in [18], the high sensitivity of TES method may come with high ratio of false positives, which should be investigated with more in-depth evaluation. In such systems, the false negatives should also be considered as one of important evaluation metrics.

The goal of this work is to make an in-depth analysis of Netflow traffic traces, and to provide the basis for an accurate and insightful analysis of the TES.

For this purpose, the result of TES on the data has been checked with the output of another base program FlowSketches/FlowExtractor² [14]. FlowSketches program employs a different measurement KL-distance [2] to detect anomalies. Two approaches are used:

¹If $q = 1$ the Tsallis entropy is equal to the Shannon entropy.

²Through the text we refer to them as FlowSketches, most of the time.

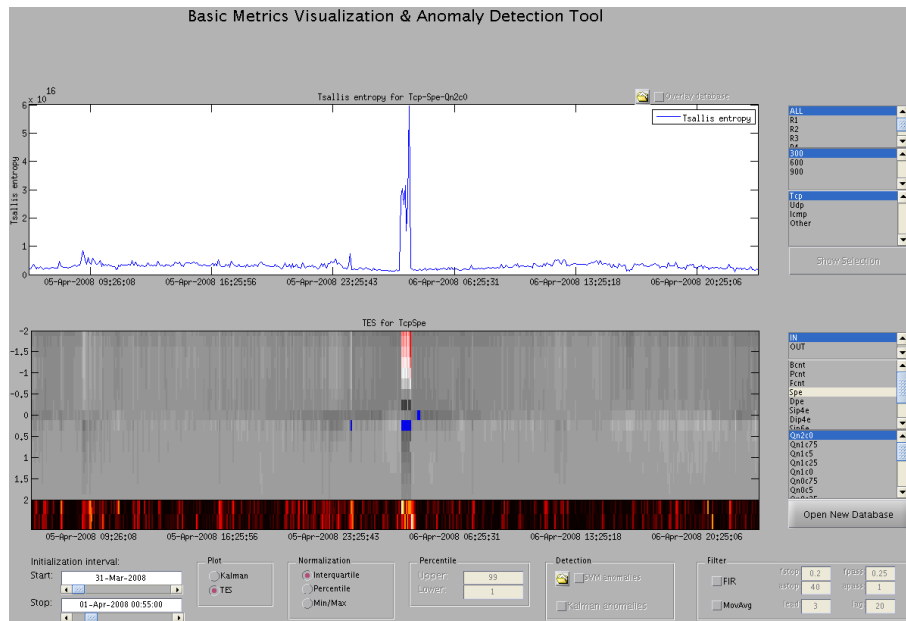


Figure 1.1: TES visualization tools.

- First: by looking at TES traces, time durations that their q Tsallis value shows abnormal behavior, are extracted. Knowing the start-time and end-time of the possible anomalies discovered by TES, we are able to cross-check this finding with the FlowSketches program: The times are set in the corresponding script to see whether the outcome of FlowSketches also indicates it as an anomaly or not. If it is, we continue the running of FlowSketches to extract the corresponding flow feature information (IP, port, etc).

In this way, we can evaluate the *false positiveness* of TES.

- Second: by running the FlowSketches on the data (that have the q Tsallis value set in one of the fields), we find out the anomalies. Then, we check q Tsallis values to see whether they are considerable based on the normal patterns or not. In case of multiple q Tsallis values, we use the arithmetic median of them in that period.

In this way, we can evaluate the *false negativeness* of TES.

In order to achieve these goals, we generally pursue the following (main) steps/contributions:

1. By using a GUI-based MATLAB tool (TES visualization), we locate time-wise the (statistically) anomalous peaks by looking at timeseries and distributions of multiple flow features (IP addresses, port numbers, flow-, packet- and byte counts).
2. By using a C++ program named FlowSketches, we automatically extract the anomalous peaks. Subsequently, by employing a sub-program FlowExtractor, we extract the feature information for every anomaly.
3. Having the outcome of previous steps, we are able to investigate the correctness of detected anomalies, part of it by human/expert involvement and other automatically.
4. Then we somehow extend the FlowSketches C++ program by adding a (stand-alone) identification program *AnalyzeTES*. This program takes the anomaly information generated in the previous steps and works on the data. It analyzes the file and puts a good possible explanation for every anomaly.
5. This program finally inserts all the information into a designed MySQL database, with a proposed data structure. The information at least contain: IP address of victim(s), IP address of attacker(s), port number of victim(s), port number of attacker(s), total size of transferred packets in this connection, time duration of that attack, start time of attack, end time of attack, q Tsallis value and finally the root cause(s) of each peak/anomaly.

Table 1.1: Information about the existing NetFlow data set

Coverage	March 2003 - today
Bytes/hour	500-2000 MB (compressed)
Total Bytes (02.2008)	approximately 38 TB (compressed)
Archive	Jabba (tape-library) Download speed: 5(script) to 10(framework) MB/s
Completeness	A few gaps or corrupt files (exact number unknown). An incomplete log already exists.

1.3.1 Data

The data used here was captured from the five border routers of the Swiss Academic and Research Network (SWITCH) [6]. SWITCH is a medium-sized operator that connects several universities and research labs to the Internet. Table 1.1 summarizes some fact about the NetFlow data set used in this work. The data are stored without any sampling or anonymization.

1.4 Overview

The rest of this report is organized as follows. Chapter 2 describes what the problem is and also brings the approaches to tackle the problem, step by step. Chapter 3 brings the database and program design of this work along with little implementation details. In Chapter 4, we see some experimental result on real SWITCH traffic data. Chapter 5 talks about outlook and some suggestions for future work. And Chapter 6 summarizes the report and gives a conclusion. Besides, in the Appendix part: Chapter A briefly mentions the implementation details of the developed program, and Chapter B brings abbreviation of terms used through the text. Later, Chapter C shows the timetable of this work, and finally Chapter D presents the original task description.

Chapter 2

Problem and Approaches

This work tries to evaluate and make an in-depth analysis of TES. In this chapter, we describe what the problem is, and how we find a solution for it.

2.1 Problem

The evaluation used for the detection of anomalies in TES are incomplete since they only focused on the exposure of a few large scale anomalies. Also the approach should be examined by several assessments. In this way, we provide the basis for an accurate and insightful analysis of the TES, specially to measure how accurate it detects anomalies. As a tool to evaluate this, we make use of:

1. Human inspection, by looking at detected anomalous traces and compare them with possible attacks.
2. Automatic inspection, by means of FlowSketches program (we will see later) to automatically find out the anomalous peaks and cross-check with TES result and vice versa.

And to measure and evaluate TES functionality, we take into account these criteria, which are the main (standard) factors to evaluate such detection systems:

1. False positive: in this context means the system detects the anomaly but it is not really an anomaly (the detection is not true).
2. False negative: in this context means the system does not detect the anomaly but it is really an anomaly (the detection is not true).

Besides these criteria, two others “true positive” and “true negative” are also very important, which in this work we implicitly consider them too but we try to focus on the two mentioned ones.

2.2 Other Available Tools: FLAME

Along with the TES and its visualization tools, we need to get familiar with FLAME as well as its sub-modules FlowExtractor and FlowSketches. FLAME (Flow-Level Anomaly Modeling Engine) [8] is a framework that is able to generate and inject realistic testing traces in order to evaluate anomaly detection systems. In this way, we have the control on the anomalous traffic and can examine the accuracy of the underlying system.

As the name speaks for itself, FLAME works at the flow level. A flow is a unidirectional series of IP packets that have the same internet protocol, the same source and destination IP addresses and port numbers, and occur at a certain period of time. A flow contains the following information, which drastically reduce the amount of data to keep for every connection:

- the source IP address,
- the destination IP address,

- the source port,
- the destination port, and
- the protocol type.

FLAME basically has these major modules:

1. NetflowReader(s): It reads flow records in Netflow (versions 5 and 9) format from flat files.
2. NetflowWriter(s): It takes as input flow records in the internal format and writes them to a flat file in Netflow (versions 5 and 9) format.
3. FlowMerger: It takes flow records in the internal format from multiple registered pipes and outputs the records to one pipe.
4. FlowDeleter: It reads flow records in internal format, calls the python script for a delete decision, and deletes the records satisfy the decision.
5. FlowGenerator: It generates request flows and outputs them in the internal format in increasing order of end time.

Lately, FlowExtrator and FlowSketches have been added to it by [14].

2.2.1 FlowExtrator

The FlowExtrator module [14] extracts flow records according to a configuration script. The script contains timing information and hash values for a specific feature. As the result of its running, FlowExtrator gives out a file containing the following feature information for every selected flow record:

- source IP address
- destination IP address
- source port
- destination port
- protocol
- number of packets
- size in bytes
- start time
- end time

2.2.2 FlowSketches

FlowSketches module/program [14] is an additional part of the FLAME framework. In this program, the flow traffic is divided into (configurable) time intervals and then is used for the generation of sketches. Afterwards, it uses the Kullback-Leibler (KL) distance [2] to compare two consecutive sketches. KL-distance is a relative entropy and can reveal the changes of network traffic, enabling us to identify the anomalous flow and its interval.

In order to filter out and identify an anomaly, [14] introduces some utility scripts that they work on the FlowSketches output and make it ready for the next step (FlowExtrator and etc). There exists a MATLAB script that uses 99% as a threshold and filters out the highest 1% of the KL-distances as anomalous traffic. A Perl script then generates an extractor configuration script that contains the necessary information for the FlowExtrator. (We will work with these scripts specially in the Chapter 4.)

Another MATLAB script removes the outliers writes the remaining records to a file. And finally, in the classification step, a scripts groups the anomalies according to their protocol, their entropies and their structural distribution.

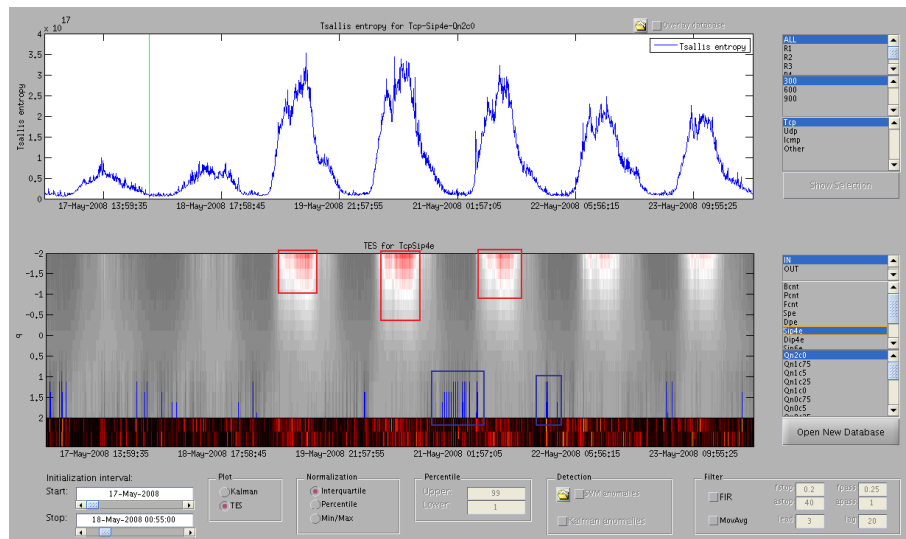


Figure 2.1: A snapshot of TES visualization: for TCP source IP, the incoming traffic (rectangles are not from TES, added for highlighting).

2.3 Approach

In order to evaluate TES, specially to measure how good it finds out the anomalous peak in a trace, we follow these two approaches:

1. Approach One (Concentrating on false positive): In this approach, we first look at the TES traces, and extract time durations that their q Tsallis value shows an anomaly pattern. We acquire and write down the start-time and end-time of the possible anomalies discovered by TES. Figure 2.1 shows a snapshot of TES visualization platform, and the possible anomalous peak/pattern.

We then cross-check this finding with the outcome of FlowSketches program at these times (as will describe later). Taking FlowSketches as our metric, we can examine the correctness of TES diagnosis, to see how many of these positive alarms (i.e. there exists an anomaly) by TES are really false. So to speak, TES was mistaken and should have NOT raised an alarm.

As the result of the first stage of this approach, we gather the start time information about the anomalous peaks from a specific data, in a certain period. In the next stages, we set this time and data information in the corresponding script (refer to section 2.3.2).

2. Approach Two (Concentrating on false negative): In this approach, we first run the FlowSketches on the data to find out the anomalies. Figure 2.2 shows a sample running and result of FlowSketches (followed by FlowExtractor) program.

Then, we extract the q Tsallis value for those times the anomalies occurred (it could be multiple values). Doing so, we are able to examine whether the q values are considerable based on the normal patterns or not. In case of multiple q values, we use the “median”¹ of them. Therefore, we can measure how many of these negative alarms (i.e. there does not exist an anomaly) are really false. So to speak, TES was mistaken and SHOULD have raised an alarm.

Instead of the median value, we could use other metrics such as: the average, the max and/or even the min among q values in that attack period. The reason we prefer the median is due to the fact that it is not (very) sensitive to outliers, the observations that are very different from all the others. While the mean as well as max/min values are affected by outliers or in some cases are outliers. A disadvantage of using median in comparison to the others is the bigger computation time it needs.

¹ The median divides the distribution into halves; half is below it, and half above it.

```

amahdi@x02: /largefs3/mahdi/refl_DDOS_april/
amahdi@x02:/largefs3/mahdi/refl_DDOS_april$ ./my_sketch.sh
amahdi@x02:/largefs3/mahdi/refl_DDOS_april$ FSK: Starting the main processing loop
/largefs1/netflow_data/raw/2008_04_12_refl_DDOS/19991_00045460_1206896400.dat.bz2
FSK: inPipe opened
FSK: outPipe opened

amahdi@x02:/largefs3/mahdi/refl_DDOS_april$ ls sketch_*
sketch_icmp_dstip_in.csv  sketch_icmp_srcip_out.csv  sketch_tcp_srcip_in.csv  sketch_udp_dstip_out.csv
sketch_icmp_dstip_out.csv  sketch_tcp_dstip_in.csv  sketch_tcp_srcip_out.csv  sketch_udp_srcip_in.csv
sketch_icmp_srcip_in.csv  sketch_tcp_dstip_out.csv  sketch_udp_dstip_in.csv  sketch_udp_srcip_out.csv
amahdi@x02:/largefs3/mahdi/refl_DDOS_april$

amahdi@x02:/largefs3/mahdi/refl_DDOS_april$ ls anomalies_*
anomalies_icmp_dstip_in_99.txt  anomalies_tcp_dstip_in_99.txt  anomalies_udp_dstip_in_99.txt
anomalies_icmp_dstip_out_99.txt  anomalies_tcp_dstip_out_99.txt  anomalies_udp_dstip_out_99.txt
anomalies_icmp_srcip_in_99.txt  anomalies_tcp_srcip_in_99.txt  anomalies_udp_srcip_in_99.txt
anomalies_icmp_srcip_out_99.txt  anomalies_tcp_srcip_out_99.txt  anomalies_udp_srcip_out_99.txt
amahdi@x02:/largefs3/mahdi/refl_DDOS_april$

amahdi@x02:/largefs3/mahdi/refl_DDOS_april$ ls analysis_tcp_hships_in_*
analysis_tcp_hships_in_1206897900.txt  analysis_tcp_hships_in_1206908700.txt
analysis_tcp_hships_in_1206898500.txt  analysis_tcp_hships_in_1206909600.txt
analysis_tcp_hships_in_1206898500_answer.txt  analysis_tcp_hships_in_1206913800.txt
analysis_tcp_hships_in_1206898800.txt  analysis_tcp_hships_in_1206914100.txt

```

Figure 2.2: Snapshot of (a) Sample running of FlowSketches, (b) Its result, (c) Result from intermediate step, (d) Outcome of FlowExtractor.

In both of these approaches, we need to extract the feature distributions (IP, port, etc) in order to point out the exact attack information that TES either could not detect or could detect properly. In the following, we describe how to do it by using the FlowSketches program step by step, which is common in two approaches.

2.3.1 AnalyzeTES program

Before we go into details of the approach, let us have a look at *AnalyzeTES* program: some of its data structures and functions that we will use in the following. Their detailed explanation will come at the next chapter (see 3).

- *anomlist* data structure: contains list of anomalies, the output of FlowSketches program.
- *attacks* data structure: keeps data about the possible attacks discovered from *anomlist*.
- *netScanAttacks* data structure: keeps the network scan attack data.
- *setTsallisEntropy* function: sets the q Tsallis value for every anomaly in *anomlist*.
- *processAnomalies* function: processes the *anomlist* and fills the *attacks* data structure.
- *analyze* function: analyzes the *attacks* to find out the attack reason, and even seeks for network scan attacks. If it could find, it fills the data into *netScanAttacks* data structure.
- *checkForNetScanAttack* function: *analyze* calls this function to check for possibility of network scan attack. If it is, this function handles the data insertion part properly.
- *postProcessAnomalies* function: Summarizing some of the fields in *attacks* and *netScanAttacks* is the responsibility of this function (specially to convert the IP addresses into their equivalent CIDR notation).

2.3.2 Finding the anomalies

We run the FlowSketches program on the pre-computed dataset files with the demanded configuration, as an example comes below (bash file sketch.sh):

```

#!/bin/sh
SKETCHDIR="/home/amahdi/fromevelyn/sketch-1.0/src"
READERDIR="/home/amahdi/work/workspace/Netflowv5Reader/Debug"
MERGERDIR="/home/amahdi/work/workspace/FlowMerger/Debug"
DATA1="/largefs1/netflow_data/raw/2008_04_12_refl_DDOS/19991*.dat.bz2"

```

```

DATA2="/largefs1/netflow_data/raw/2008_04_12_refl_DDoS/19993*.dat.bz2"
START=1208108700
INT=300
LEN=1024
HSH=5
#####
# Create pipes
mkfifo pipe1
mkfifo pipe2
mkfifo pipe3
$READERDIR/Netflowv5Reader $DATA1 pipe1 &
$READERDIR/Netflowv5Reader $DATA2 pipe2 &
$SKETCHDIR/FlowSketches pipe3 /dev/null -h $HSH -l $LEN -i $INT \
                                -s $START &

$MERGERDIR/FlowMerger pipe3 pipe2 pipe1
# Delete pipes
rm pipe1
rm pipe2
rm pipe3

```

The output of FlowSketches are files with “sketch_” as prefix for every feature, for example:

```

sketch_icmp_dstip_in_1.csv, sketch_tcp_dstip_in_1.csv,
sketch_udp_dstip_in_1.csv, sketch_icmp_dstip_out_1.csv,
sketch_tcp_dstip_out_1.csv, sketch_udp_dstip_out_1.csv,
sketch_icmp_srcip_in_1.csv, sketch_tcp_srcip_in_1.csv,
sketch_udp_srcip_in_1.csv, sketch_icmp_srcip_out_1.csv,
sketch_tcp_srcip_out_1.csv, sketch_udp_srcip_out_1.csv

```

Each of these files has the following information:

- Init-values for 5 hash functions (in the above example, $HSH = 5$)
- Number of records falling into each of 1024 bins (in the above example, $LEN = 1024$), each with 5 hash functions (so in total: $1024 = 5120$ records)
- KL-distances in forward and backward direction for two consecutive sketches

After this step, collect_anomalies.m MATLAB program is run on the output of previous step. This MATLAB script extracts the single-bin anomalies and outputs a separate file for every feature (having the “anomalies_” as prefix), with these information:

- The init-values for the hash functions,
- Followed by a list of collected anomalies, which every line contains the anomaly's start time, the interval number and the hash values for the different hash functions.

Sample output of collect_anomalies.m program is:

```

1804289383 846930886 1681692777 1714636915 1957747793
1187557500 1 801 589 267 949 800 0.564833 0.466312
1187558100 3 999 371 723 1005 39 0.354194 0.18411
1187558400 4 47 577 52 930 303 0.662444 0.265222

```

In the sequel, the generate.pl script is executed (Usage: *perlgenerate.pl*). It reads the anomalies and then generates configuration script extract.sh (along with the Python script analysis1.py), which is something like the following:

```

#!/bin/sh
SKETCHDIR="/home/amahdi/fromevelyn/sketch-1.0/src"
READERDIR="/home/amahdi/work/workspace/Netflowv5Reader/Debug"
# Create pipes
mkfifo pipe1

```

```
# Start programs
$READERDIR/Netflowv5Reader /largefs1/netflow_data/raw/ \
2007_08_31_epfl_ddos/19991_00040078_1187557200.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040079_1187560800.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040080_1187564400.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040081_1187568000.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040082_1187571600.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040083_1187575200.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040084_1187578800.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040085_1187582400.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040086_1187586000.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040087_1187589600.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040088_1187593200.dat.bz2 \
/largefs1/netflow_data/raw/2007_08_31_epfl_ddos/ \
19991_00040089_1187596800.dat.bz2
$SKETCHDIR/FlowExtractor pipel /dev/null analysis1.py
# Delete pipes
rm pipel
```

After running the `extract.sh` script, the output of FlowExtractor are text files with “analysis_” as prefix. Every file contains several records, each of them with the following information (as mentioned before): source IP address, destination IP address, source port, destination port, protocol, number of packets, size in bytes, start time, and end time.

Following is a sample output of FlowExtractor program (for privacy purposes, the IP addresses are anonymized):

```
3422584391 2164804241 8 0 1 1 64 1187557192734 1187557192734
1782954186 2164804241 8 0 1 1 64 1187557196968 1187557196968
2490984179 2164804241 8 0 1 1 64 1187557196960 1187557196960
1746457941 2164804241 8 0 1 1 64 1187557200677 1187557200677
```

We then run the `remove_outliers.m` MATLAB script on the outcome to remove outliers if any.

2.3.3 Pre-processing

Having the output of FlowExtractor program (files with prefix “analysis_”), the program initiates as follows:

1. It first reads the files and stores them in appropriate data structure (*anomalist*).
2. Then it cross-checks every timestamp in the anomalies with TES database, maintaining the corresponding Tsallis q values from the pre-computed .db files (by calling *setTsallisEntropy* function).
3. It computes and hereafter uses the median q among the extracted q Tsallis values in the duration of start time and end time of that anomaly.

The program then goes through analysis on the data. Before going into attack identification, it should pre-process the data as such it categorizes every anomaly information based on its destination IP address. This IP actually represents the *victim*, which is the principal part of every attack. At this stage, the program summarizes the information like the following:

Table 2.1: Considered attacks, part one.

Attack	Description
DOS	Denial of Service Attack (distributed or single-source) which affects destination address, source address.
ICMP Flooding	Consuming the victim's bandwidth by sending a huge amount of echo ICMP requests.
Reflector - SMURF	Sending broadcast ICMP packets to a whole subnet, so that all (or at least most) of the machines in that subnet then send a reply to the victim. Solution: blocking ICMP broadcast at the router.
TCP SYN Flooding	Exploiting the three-way handshake part of the TCP protocol by sending many SYN packets to one host. For detection: the number of TCP flows, average number of packets in each TCP flow, average number of bytes in each TCP flow, number of unique IP addresses seen (all per minute) can be used. Solution: SYN cookies
UDP Flooding	Sending a very large number of UDP packets to one or several random ports of a victim. These packets will eventually consume all of the available bandwidth and thus lead to a Denial of Service (DoS). Detection: the number of UDP flows per minute is analyzed.
DNS Reflector	Sending a flood of DNS requests with a spoofed IP address (the one of the victim) to one or more DNS servers, which results in a flood of DNS responses sent to the victim. Detection: Checking the high rate of DNS requests from the same (spoofed) IP address to a DNS server inside the network, or an unusual high number of UDP flows with source port 53 (of DNS)

- Group of attackers to this victim are being put together.
- The total number of packets in this anomaly are summed up.
- The total packets size of this anomaly is computed.
- The duration of this anomaly is calculated as well.

2.3.4 Analysis

In the analysis phase (mainly in *processAnomalies* and *analyze* functions), it checks for different well-known anomalies on the summarized data. It considers the attacks listed in Tables 2.1 and 2.2, all taken from [13, 18].

For the attacks to be accurately identified, we should define appropriate thresholds, accordingly. The thresholds are highly dependent on the type of underlying network (i.e. small, medium, big), a specific time (i.e. high season, low season), and a lot of other factors. Mostly, using an outlier based approach and tuning the values by experiment give us a good outcome. Although coming up with a constant and general threshold is not an easy job, we realize that the following values somehow well fit with our medium level network²:

- *noIpsThreshold* = 20×50 : Threshold for the maximum number/size of attacker IP addresses to store in the fields. The value $20 \times 50 = 1000$ means 50 IPs each of which with size 20 characters (i.e. representation of IPs in string dotted format with a comma as delimiter).

²All of the thresholds are configurable and could be changed in the program.

Table 2.2: Considered attacks, part two.

Attack	Description
Blaster Worm	Exploiting an RPC (Remote Procedure Call) vulnerability on TCP destination port 135.
Witty Worm	Exploiting a vulnerability in ISS network security products, using UDP source port 4000 and random destination port.
Alpha Flows	Unusually large volume point to point flow. Source address and destination address (possibly ports) are affected in this attack.
Flash Crowd	Unusual burst of traffic to single destination, from a distribution of sources. Destination address, destination port are affected.
Port Scan	Probing many destination ports on a small set of destination addresses.
Network Scan	Probing many destination addresses on a small set of destination ports. Destination address, destination port are affected.
Outage Events	Traffic shifts due to equipment failures or maintenance. Mainly source and destination address are affected.
Point to Multi-point	Traffic from single source to many destinations. Source address, destination address are affected.
DDoS 1 [18]	A short (10 minutes) DDoS attack on a router and a host with 8 million spoofed source addresses. Destination port is TCP 80.
DDoS 2 [18]	A long (13 hours) DDoS attack on a host with 5 million spoofed source addresses. Destination port is TCP 80.

- $portScanThreshold = 4 \times 50$: Threshold for the maximum number of port scans to be counted as Port Scan attack. The $4 \times 50 = 200$ is the average size in string: at minimum, if an attacker tries 50 different ports each with length 4 in string format.
- $longDurationThreshold = 10 \times 60 \times 60 \times 1000$: Threshold for the maximum duration of an anomaly for a special kind of attack. We put a limit of almost $10 \times 60 \times 60 \times 1000 = 10h$ (hours) in this respect.
- $trafficThreshold = 500 \times 2048$: Threshold for the maximum traffic between the attackers and a victim. This value is in bytes: 500 packets each of which having size in average 2 Kbyte.
- $ipScanningThreshold = 100$: Threshold for the number of network scanning made by an attacker to be counted as Network Scan attack. It seems that 100 IPs per attacker be a reasonable threshold.

Having the above mentioned thresholds, we identify the attacks as follows (based on the knowledge given in Tables 2.1 and 2.2):

- Alpha Flows: it checks if it is one attacker in the entry (of the *attacks* data structure) and the traffic between the attacker and the victim is more than the threshold (i.e. the *trafficThreshold*).
- Flash Crowd: it checks if it is an ICMP entry, just one single attacker IP and port, and the total size of traffic exceeds the *trafficThreshold*.
- ICMP Flooding: if it is an ICMP incoming entry.
- ICMP Reflection: if it is an ICMP outgoing entry.
- Blaster Worm: if it is a TCP incoming entry from attacker to victim port of 135.

- DDoS 2 (A long DDoS): if it is a TCP incoming entry with victim port 80, and the duration of attack is more than the *longDurationThreshold*.
- DDoS 1 (A short DDoS): the same as above, but without fulfilling the threshold.
- Port Scan: if it is a TCP incoming entry with number of victim ports more than the *portScanThreshold*.
- TCP Reflection: if it is a TCP outgoing entry.
- UDP Flooding: if it is an ICMP incoming entry (as a general detection in this case).
- Witty Worm: if it is an ICMP incoming entry, with attacker port 4000.
- DNS Reflector: if it is an ICMP outgoing entry, with attacker port 53 (for DNS).
- UDP Reflection: if it is an ICMP outgoing entry and not a DNS reflector attack, it is most probably a UDP Reflection attack.

Based on this sort of checking, it then puts a good possible explanation as the “reason” field of the *attacks* data structure. Also if it is a network scan attack (as the result of calling *checkForNetScanAttack* function), the necessary data are filled into the *netScanAttacks* data structure as well. Detecting network scan attack is a bit different, the next section argues about it.

Network Scan Attack

In network scan attack usually one attacker scans a range of victims or so to speak IP addresses. So if we process as before (grouping based on victim IP address), it is hard to observe anomalous traffic since each of the victims just receives one request/probe from the attacker and nothing more. But if we categorize them from the attacker’s point of view and gather all of the victims in one field and attacker in another field, this huge number of (scanned) victims can reveal anomalous behavior.

Therefore, in order to check the possibility of network scan attack we proceed as follows: We put attacker in one side and gathers all of the victims in another side, all in *netScanAttacks* data structure (reverse of what we do in other attacks).

2.3.5 Post-processing

Before releasing the result, a post-processing phase is performed by calling *postProcessAnomalies* function. Since the attack field of *attacks* data structure may contain a large number of (sequential) IP addresses, a good idea is to summarize them in CIDR notation [10]. In this way, for example, all of the 256 IP addresses from 192.168.0.0 to 192.168.0.255 can be re-written in condensed form of “192.168.0.0/24”.

2.3.6 Writing the result

The program then inserts the detected attacks and list of anomalies into:

1. The output script files: it generates scripts based on MySQL standard definition, which later on can be executed on a real database (providing facile import/export from/to database).
2. The MySQL database: it instantly connects to a database and inserts the data into the designed tables.

Chapter 3 brings more details.

2.4 Summing up

We first locate the time and duration of anomalous peaks in TES visualization tools for a specific time series. By a program, we also extract the corresponding q Tsallis values for detailed view on the anomalies. Then by running FlowSketches on the raw traffic data, we automatically extract the anomalous peaks from the specific time series or even from other time points. Subsequently, by feeding the output to FlowExtractor program with correct parameters, we take out the feature information (including source IP address, destination IP address, and so on) for every anomaly. We consider the result of this step as our base metrics to evaluate and examine TES output.

The C++ program AnalyzeTES is then run on the FlowSketches/FlowExtractor output and it identifies the attacks occurred in the anomalous peaks. The detected attacks are investigated based on their TES pattern (as well as q Tsallis values) to see whether TES marked this one as attack or not (if not, it means false negative). Also the other way around, a detected anomalous peak in TES is also cross-checked with the detected attacks and if it is not among them, so it means false positive by TES. Of course, expert investigation is also crucial in some conflicts, due to the fact that FlowSketches output could not be taken as a one-hundred percent true result.

In order to store data in an efficient manner, we design a MySQL database, including various tables and necessary relations. Again the AnalyzeTES is in charge of maintaining the database, and when the result is ready, is in charge of inserting all the detailed data into the corresponding tables. We try to develop all capabilities in just one programming language and one stand-alone program, avoiding the need for different programs to be installed or managed. The information in the database tell almost everything about an anomaly and attack, from IP address of victim(s), IP address of attacker(s), port number of victim(s), port number of attacker(s), total size of transferred packets, time duration of that attack, start time of attack, end time of attack, q Tsallis value and finally the root cause(s) of each peak/anomaly.

Chapter 3

Design and Implementation

In this chapter, we bring the database design as well as program design. The program design section briefly introduces main points of C++ implementation, and leaves the details for the Appendix A.

3.1 Database Design

In order to store the (raw) anomaly data, the (processed) discovered attacks, and related information, the standard way is to use Database Management Systems (DBMS). Afterwards, we simply query the database, and we can extract the knowledge in a very efficient and fast manner. Accordingly, we choose MySQL DBMS [4] and design at least 4 tables as follows. Figure 3.1 depicts an overall view of tables in MySQL Workbench 5.1¹ software.

3.1.1 Anomaly database table

We need one table to store the unprocessed anomalies for further queries. Definition of *Anomaly* table in MySQL format comes below:

```
CREATE TABLE IF NOT EXISTS `mydb`.`Anomaly` (  
  `idAnomaly` INT NOT NULL AUTO_INCREMENT ,  
  `srcip` CHAR(15) NOT NULL ,  
  `dstip` CHAR(15) NOT NULL ,  
  `srcport` INT NOT NULL ,  
  `dstport` INT NOT NULL ,  
  `proto` INT NOT NULL ,  
  `packets` INT NOT NULL ,  
  `size` INT NOT NULL ,  
  `stime` TIMESTAMP NOT NULL ,  
  `etime` TIMESTAMP NOT NULL ,  
  `q` DOUBLE NULL ,  
  PRIMARY KEY (`idAnomaly`) );
```

This table contains these columns for each anomaly, respectively:

- idAnomaly: an auto incremental number as the primary key of this table
- srcip: source IP address of this anomaly
- dstip: destination IP address
- srcport: source port number
- dstport: destination port number
- proto: the standard protocol number (e.g. 6 for TCP protocol)

¹ URL: <http://wb.mysql.com/>

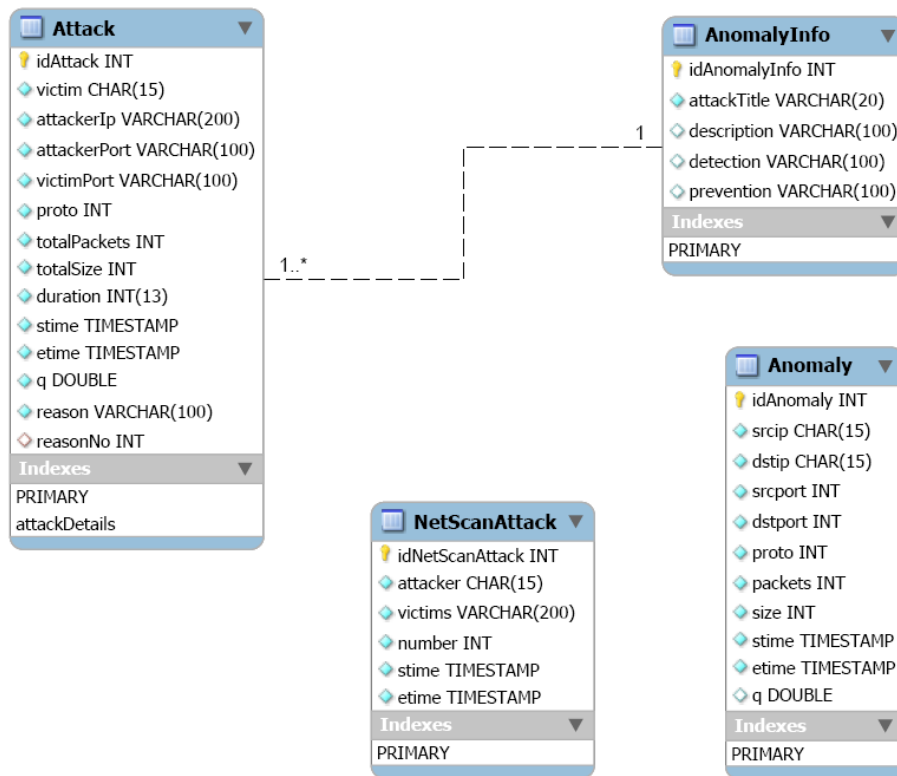


Figure 3.1: Database design: relation and fields of tables.

- packets: number of packets
- size: the size of traffic data in byte
- stime: start time of that anomaly
- etime: end time of that anomaly
- q: q Tsallis value of the anomaly

3.1.2 Attack database table

We summarize the processed and discovered attacks in another table, *Attack*. Definition of *Attack* table in MySQL comes below:

```
CREATE TABLE IF NOT EXISTS `mydb`.`Attack` (
  `idAttack` INT NOT NULL AUTO_INCREMENT ,
  `victim` CHAR(15) NOT NULL ,
  `attackerIp` VARCHAR(200) NOT NULL ,
  `attackerPort` VARCHAR(100) NOT NULL ,
  `victimPort` VARCHAR(100) NOT NULL ,
  `proto` INT NOT NULL ,
  `totalPackets` INT NOT NULL ,
  `totalSize` INT NOT NULL ,
  `duration` INT(13) NOT NULL ,
  `stime` TIMESTAMP NOT NULL ,
  `etime` TIMESTAMP NOT NULL ,
  `q` DOUBLE NOT NULL ,
  `reason` VARCHAR(100) NOT NULL ,
  `reasonNo` INT NULL ,
  PRIMARY KEY (`idAttack`) ,
  INDEX `attackDetails` (`reasonNo` ASC) ,
```

```

CONSTRAINT `attackDetails`
  FOREIGN KEY (`reasonNo` )
  REFERENCES `mydb`.`AnomalyInfo` (`idAnomalyInfo` )
  ON DELETE NO ACTION
  ON UPDATE NO ACTION);

```

The columns for every attack are, respectively:

- idAttack: an auto incremental number as the primary key of Attack table
- victim: IP address of victim
- attackerIp: IP address(es) of attacker(s), which may be represented in CIDR format (if they are serial)
- attackerPort: port number(s) of attacker(s)
- victimPort: port number of victims
- proto: the protocol number (TCP, UDP, ICMP)
- totalPackets: the total number of packets transmitted in this attack
- totalSize: the total size in byte of the traffic data
- duration: the duration period of this attack
- stime: start time of attack
- etime: end time of attack
- reason: an explanation about the type of this attack
- reasonNo: reason number, which refers to the corresponding row in *AttacksInfo* table (its definition comes later)

3.1.3 NetScanAttack database table

As mentioned before, the network scan attack is a bit different from other attacks and hence we have to create a new table for them, named *NetScanAttack*. It is worth mentioning that this table is a replication of *Attack* table; means the corresponding data of this table is also available in the *Attack* table (with different fields, of course). Although we could simply remove them in the insertion phase, we tend to keep them for further analysis and double checking, if necessary. Definition of *NetScanAttack* table in MySQL is as follows:

```

CREATE TABLE IF NOT EXISTS `mydb`.`NetScanAttack` (
  `idNetScanAttack` INT NOT NULL AUTO_INCREMENT ,
  `attacker` CHAR(15) NOT NULL ,
  `victims` VARCHAR(200) NOT NULL ,
  `number` INT NOT NULL ,
  `stime` TIMESTAMP NOT NULL ,
  `etime` TIMESTAMP NOT NULL ,
  PRIMARY KEY (`idNetScanAttack`) );

```

In this table, the following columns for every network scan attack are defined (respectively):

- idNetScanAttack: an auto incremental number as the primary key of NetScanAttack table
- attacker: IP address of attacker
- victims: IP addresses of victims, which may be represented in CIDR format (if they are serial)
- number: number of victims
- stime: start time of attack
- etime: end time of attack

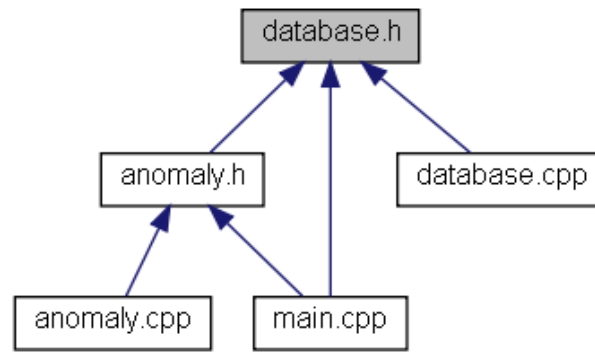


Figure 3.2: Class dependency.

3.1.4 AttacksInfo database table

We summarize the attack information (as also mentioned in tables 2.1 and 2.2) in the table *AttacksInfo*. It helps the user to figure out how to tackle and even battle the attack, for instance by looking at the prevention field. The data could be updated gradually if new attacks or new countermeasures are found.

Definition of *AttacksInfo* table in MySQL format is as follows:

```

CREATE TABLE IF NOT EXISTS `mydb`.`AnomalyInfo` (
  `idAnomalyInfo` INT NOT NULL AUTO_INCREMENT ,
  `attackTitle` VARCHAR(20) NOT NULL ,
  `description` VARCHAR(100) NULL ,
  `detection` VARCHAR(100) NULL ,
  `prevention` VARCHAR(100) NULL ,
  PRIMARY KEY (`idAnomalyInfo`) );
  
```

And respectively, the columns have these meanings:

- `idAnomalyInfo`: an auto incremental number as the primary key of *AttacksInfo* table
- `attackTitle`: a title for every attack, e.g. 'Alpha Flows'
- `description`: a (concise) description of that attack, e.g. 'large volume point to point flow'
- `detection`: describes how to detect this kind of attack
- `prevention`: describes how to prevent the attack, even pointing out to the related security patches.

3.2 Program design

AnalyzeTES has been developed to mainly fulfil these requirements: 1-to process the anomalies and discover well-known attacks out of them, 2-to store the outcome into the designed databases. All the programming code has been implemented in C++, even the interface for the database part (using MySQL C API [3]). Doing this way is more efficient and easier to use, integrate, and maintain.

The program contains these classes: *Anomaly*, *Database* and *Utility*, each of which handles the different major part of the *AnalyzeTES*. Figure 3.2 shows the dependency of these classes. In the following we just introduce some of the functions and their functionality. The more detailed information are available in Appendix A.

3.2.1 Main function

As always, the C++ code starts with a *main* function. It first validates the input arguments and if they are correctly feeded (we will see the input arguments in section Usage 3.3.2), then calls the appropriate functions (with the right settings), accordingly.

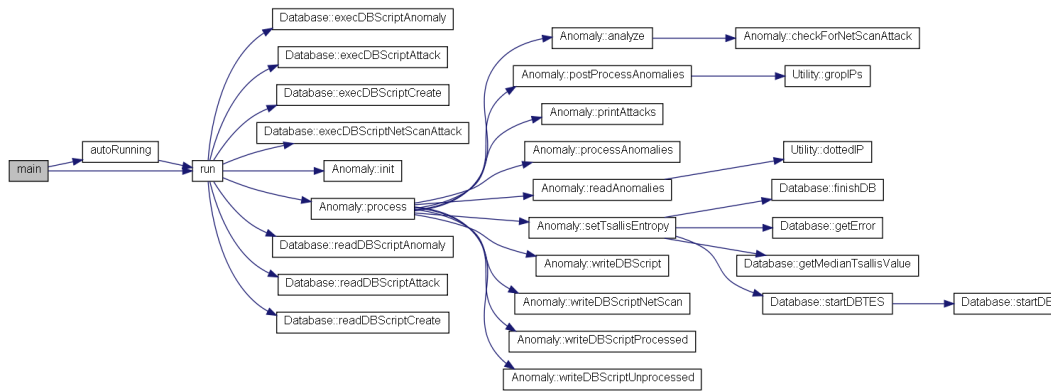


Figure 3.3: Call graph of main function.

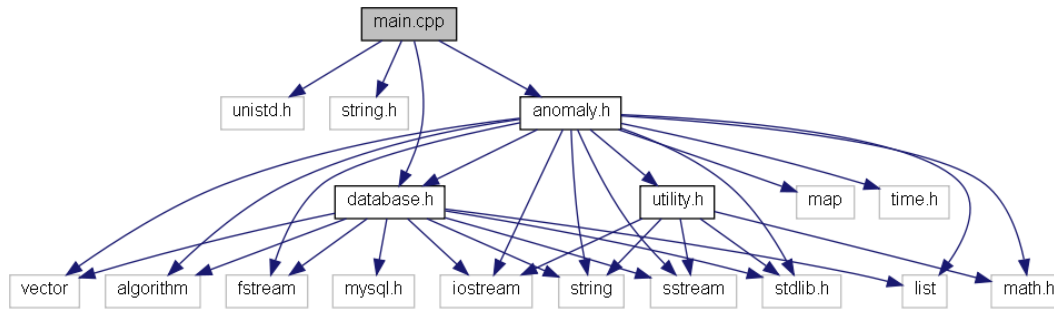


Figure 3.4: Dependency graph of main function.

Figure 3.3 sketches the call graph of *main* function: It calls either *run* or *autoRunning* function, and the procedure goes on (mainly) with *process* function from Anomaly class, that we will see later on.

The dependency graph of main function is depicted in Figure 3.4, indicating which files (and hence libraries) it uses.

3.2.2 Anomaly class

Anomaly class contains data structures and functions to load anomaly outputs from FlowSketches/FlowExtractor program, and process and analyze the data to identify possible attacks. Figure 3.5 represents the dependency of this class to other classes in the program.

For usage of this class, we should call *init* and *process* functions as below:

```
#include "anomaly.h"
Anomaly anom;

//Just for testing purposes, and not necessary.
```

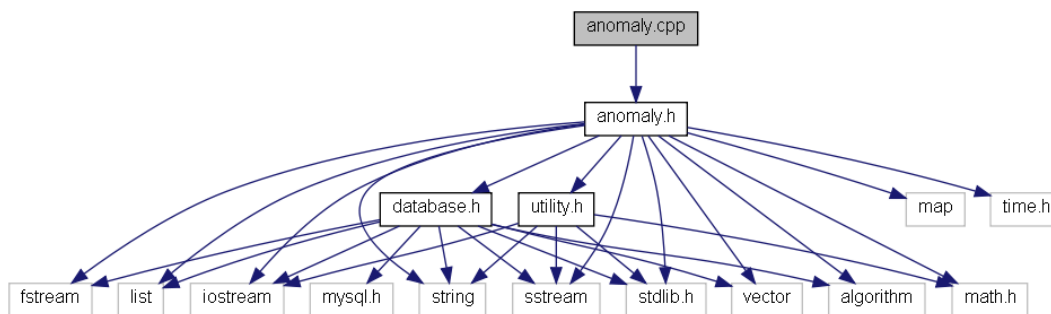


Figure 3.5: Dependency graph of Anomaly class.

```

anom.test();

//Initializes some global and constant values,
// like thresholds and inside IP address
anom.init();
//OR Initializes the input filename to process, and whether
// it is incoming or outgoing traffic.
anom.init(filename, incoming);

//Calls all of the process from scratch, from reading the
// files to the attack identification and writing back to the database.
anom.process();

```

Briefly speaking:

- The manager is the *process* function.
- As input, it gets the output file of FlowSketches and cross-check every timestamp with the TES database in order to extract q Tsallis values from the pre-computed .db files (*setTsallisEntropy* function). It computes and hereafter uses the median q in the duration of start time and end time of that anomaly.
- It then goes through analysis on the data:
 1. First it categorizes, based on the destination IP (victim), gathers every attacker, sums up the total packets and sizes, sets duration of anomalies and the like (*processAnomalies* function).
 2. Subsequently, it checks for different well-known anomalies on the data and puts a good possible explanation as the “reason” field (*analyze* function and also *checkForNetScanAttack* function for Network Scan attack).
 3. Finally, it inserts the data (detected attacks and so on) into the database or the output script file (configurable).

The mentioned functions work on the following data structures.

Info struct

Info is a data structure for keeping the raw anomaly list, acquired in previous phases. It contains all information the previous steps have generated so far, in a more comprehensive and human-understandable format:

```

struct Info {
    string srcip;
    string dstip;
    uint16_t srcport;
    uint16_t dstport;
    uint16_t proto;
    uint16_t packets;
    uint16_t size;
    uint64_t stime;
    uint64_t etime;
    double q;
};

```

In this structure: *srcip* stands for source IP address and *dstip* for destination IP address, all converted into dotted-notation format. *srcport* and *dstport* are source and destination ports, respectively. *proto* represents the protocol number, which is one of the following constant values:

```
enum PROTO {TCP = 6, UDP = 17, ICMP = 1};
```

packets shows the number of packets and *size* field shows the size in bytes. *stime* and *etime* respectively stands for start time and end time of that anomaly.

Finally, the q field here refers to “median” q Tsallis value at time period of *stime* to *etime*.

anomlist is a Standard Template Library (STL) *list* of *Info* data structure.

ProcessedInfo struct

ProcessedInfo is a data structure for keeping all the processed data: after the *anomlist* is processed, the knowledge extracted from it stores here. It contains the information about the attack, in particular the attackers IP addresses, and some aggregated information about the feature distributions.

```
struct ProcessedInfo {
    string attackerIp;
    string attackerPort;
    string victimPort;
    uint16_t proto;
    int totalPackets;
    uint32_t totalSize;
    uint64_t duration;
    uint64_t stime;
    uint64_t etime;
    double q;
    string reason;
};
```

In this data structure: *attackerIP* contains a list of attackers' IP addresses to a particular victim (the IP address of this victim is not stored here, but in *attacks* variable) in dotted-notation, separated by a comma. *attackerPort* is the list of attackers' port numbers, separated by comma as well. *proto* is one of these standard protocol numbers: TCP = 6, UDP = 17, ICMP = 1. *totalPackets* represents the total number of packets from the group of attackers to this victim, and *totalSize* is the total size in bytes of those packets. *duration* shows the time period that this attack happened, and *stime* and *etime* are start time and end time of that attack, respectively. *q* is the median *q* Tsallis value in this period, and finally *reason* contains an explanation about the type of this attack.

attacks variable is a STL map of this *ProcessedInfo* structure along with the victim in string dotted-notation format. In this way, both the victim and the attack's information are being kept together.

NetScanAttackNode struct

NetScanAttackNode is a data structure for keeping the network scan attack data to put together the attacker and its target victims.

```
struct NetScanAttackNode {
    string victims;
    int n;
    uint64_t stime;
    uint64_t etime;
};
```

In this data structure: *victims* field contains a list of victims' IP addresses targeted by a particular attacker (which the IP address of this victim is not stored here, but in *netScanAttacks* variable) in dotted-notation, separated by a comma (which later will be converted into CIDR notation, if possible). *n* is the number of all victims, *stime* and *etime* are start time and end time of that network scan attack, respectively.

netScanAttacks variable is a STL map of this *NetScanAttackNode* structure along with the attacker IP address in string dotted format. In this way, both the attacker and the victims' information are being kept together.

KnownAttackNode struct

Along with the data structures mentioned so far, for the sake of simplicity, just the victim and the attackers and the reason of this attack are also stored in *KnownAttackNode* structure as defined below.

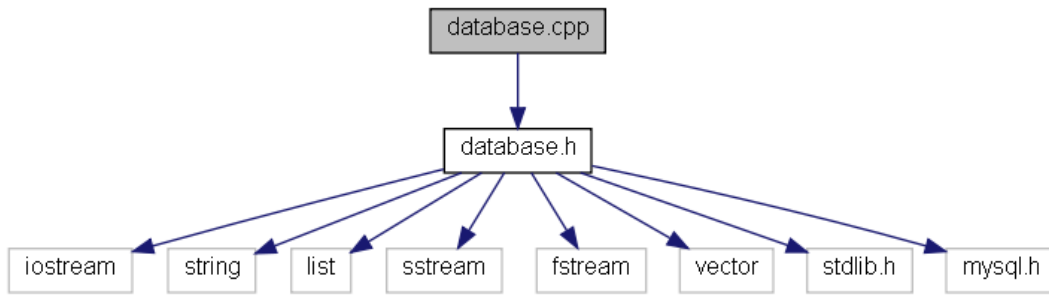


Figure 3.6: Dependency graph of Database class.

```

struct KnownAttackNode {
    string reason;
    string victim;
    string attackers;
};
  
```

Actually this data structure summarizes the findings, and makes easier to address and refer to them whenever it is needed.

Here *reason* means the identified type of this attack, *victim* contains the victim's IP address, and the *attackers* is the list of attackers to this particular victim.

3.2.3 Database class

Database class contains data structures and functions to connect to a MySQL [4] database and executes the following tasks:

1. Creating tables for the list of anomalies and the discovered attacks.
2. Inserting anomalies into *anomaly* table, and inserting attacks into *attack* table.
3. Responding different queries about the stored data.

Database class handles all considerations of working with a MySQL database, and eases its calling and error checking. Figure 3.6 sketches the dependency graph of this class.

For usage, we can call for instance the *readDBScriptCreate* and then *execDBScriptCreate* as below:

```

#include "database.h"
Database db;
//reading information from the input file 'table.txt'
db.readDBScriptCreate("tables.txt");
//creating the tables by executing the script taken from file
// to database
db.execDBScriptCreate();
  
```

Briefly speaking, it contains some function to read the data that should be stored in a database from input files, such as: *readDBScriptAnomaly*, *readDBScriptAttack*, and *readDBScriptCreate*. And some to execute a query on the database, mainly in these functions: *execDBScriptAnomaly*, *execDBScriptAttack*, and *execDBScriptCreate*.

3.2.4 Utility class

Utility class contains utility functions to work with IP addresses, port numbers, and timestamps:

1. Converting string values to integers,
2. Converting time in epoch to human readable, and
3. Converting a range of IP addresses into CIDR notation.

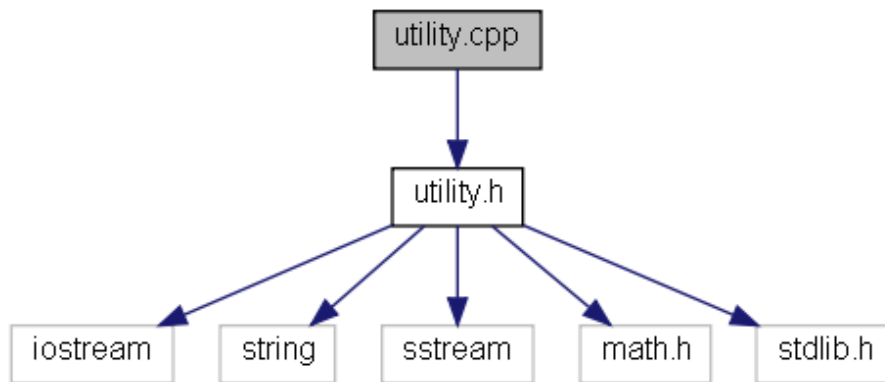


Figure 3.7: Dependency graph of Utility class.

Mostly, it is used by Anomaly class (see section 3.2.2). Figure 3.6 sketches the dependency graph of this class.

For usage, we can call it like the following:

```
#include "utility.h"

Utility util;
```

Briefly speaking, this is a utility class specially to make shorten the length of a group of IP addresses: *groupIPs* function represents a range of IP addresses in a CIDR format, e.g. $100.100.100.101 - 100.100.100.255 \Rightarrow 100.100.100.100/24$ and etc.

3.2.5 Base flowchart and summing up

As also depicted in Figure 3.8, the basic flow of AnalyzeTES program is as follows (the items before “::” represent the classes that that subsequent functionality is part of):

1. Main::process input parameters: In this phase, the main program processes the input parameters and based on them decides to call the anomaly class or not.
2. Anomaly::initialize and read anomalies: This phase deals with initializing variables like the input file and if the traffic is incoming or outgoing. Then, it reads the anomalies and converts some of the fields like IP addresses into more readable format.
3. Utility::convert IP address to dotted format: By calling this part, an IP address is converted into more comfortable format for humans, so to speak in dotted style, in order to ease the analysis part.
4. Database::connect to MySQL database: After reading the anomalies by Anomaly class, it needs to find out the corresponding *q* Tsallis values for every one of them. So in this phase, it makes a connection to the Tsallis pre-computed database.
5. Database::read *q* Tsallis values: From the last phase, it continues with reading of the demanded information from the database, of course if the connection was obtained successfully.
6. Anomaly::pre-process anomalies: In order to organize the anomalies into a victim-oriented format, this phase works on them and categorizes/groups by them based on the same victim IP address. In this way, it concentrates on the victim as the reason/target of every possible attack.
7. Anomaly::analyze them on possible attacks: This phase cross-checks the anomalies with the list of known attacks in order to identify them based on the attacks’ characteristics.

8. Anomaly::check the possibility of network scan attack: Also, since the nature of network scan attack is a bit different and can not easily be viewed in the victim-oriented format (usually an attacker scans a vast range of IP addresses to gather information about possible victims for later intended attack), it goes through an attacker-oriented one. In other words, it now categorizes the anomaly based on its attacker IP address. If the range of victim IP addresses exceeds a pre-set threshold value so it is (possibly) a network scan attack.
9. Anomaly::post-process the attacks, summarize them in CIDR format: After the analysis phase, this phase summarizes some of large fields like attackers (or victims) IP addresses in detected attacks (or respectively network scan attacks) in equivalent compact CIDR format. (Specially for the network scan attacks, due to the fact that their victims' IP addresses usually are serially ordered.)
10. Utility::show in CIDR format: It converts the range of IP addresses into the equivalent CIDR notation, if possible. For example, instead of having 256 IP addresses from 192.168.0.0 to 192.168.0.255, just one "192.168.0.1/24" is considered and stored.
11. Anomaly::write the database scripts into output files: This phase deals with writing the anomalies, the result of analysis including the attacks and so on into the script files in MySQL language. These scripts can be executed later on on every machine equipped with MySQL DBMS, and actually as a kind of exporting data for future import.
12. Main::make ready data for database: Since the script files have been created and filled, this phase passes them by calling the database related functions on each of them, in correct sequence: The first script should be table creation (if not exist) followed by a bunch of insertions.
13. Database::connect to MySQL database and create the tables if do not exist: At the first step of every database activity, it should acquire a handle to a running MySQL database. If this phase is invoked without error, it means that a (back-end) DBMS is ready to receive queries and respond accordingly. Needless to say that the necessary tables should be created first and then the data be inserted.
14. Database::insert data into the tables: Finally, the gathered data are inserted into corresponding tables for further queries.

3.3 Installing and Usage the program

3.3.1 Install

There is no installation needed for AnalyzeTES program. Since it has been developed in pure C++, it can be compiled and built in both Windows and Linux operating systems (binary files for each of these platforms are available as well). The only necessary library (other than standard C++ libraries) is the MySQL C API [3], that also the same for both operating systems.

This is the *Makefile* script of how to build the AnalyzeTES:

```
CXX=g++
CXXFLAGS=
PROG=analyzeTES
INCLUDE=mysql/include/
LIBDIR=mysql/lib/
MYSQLLIB=mysqlclient

all: main.cpp anomaly.o database.o utility.o
    $(CXX) main.cpp anomaly.o database.o utility.o \
        -l$(MYSQLLIB) -I$(INCLUDE) -o $(PROG)
anomaly.o: anomaly.cpp anomaly.h
    $(CXX) -c anomaly.cpp -I$(INCLUDE)
```

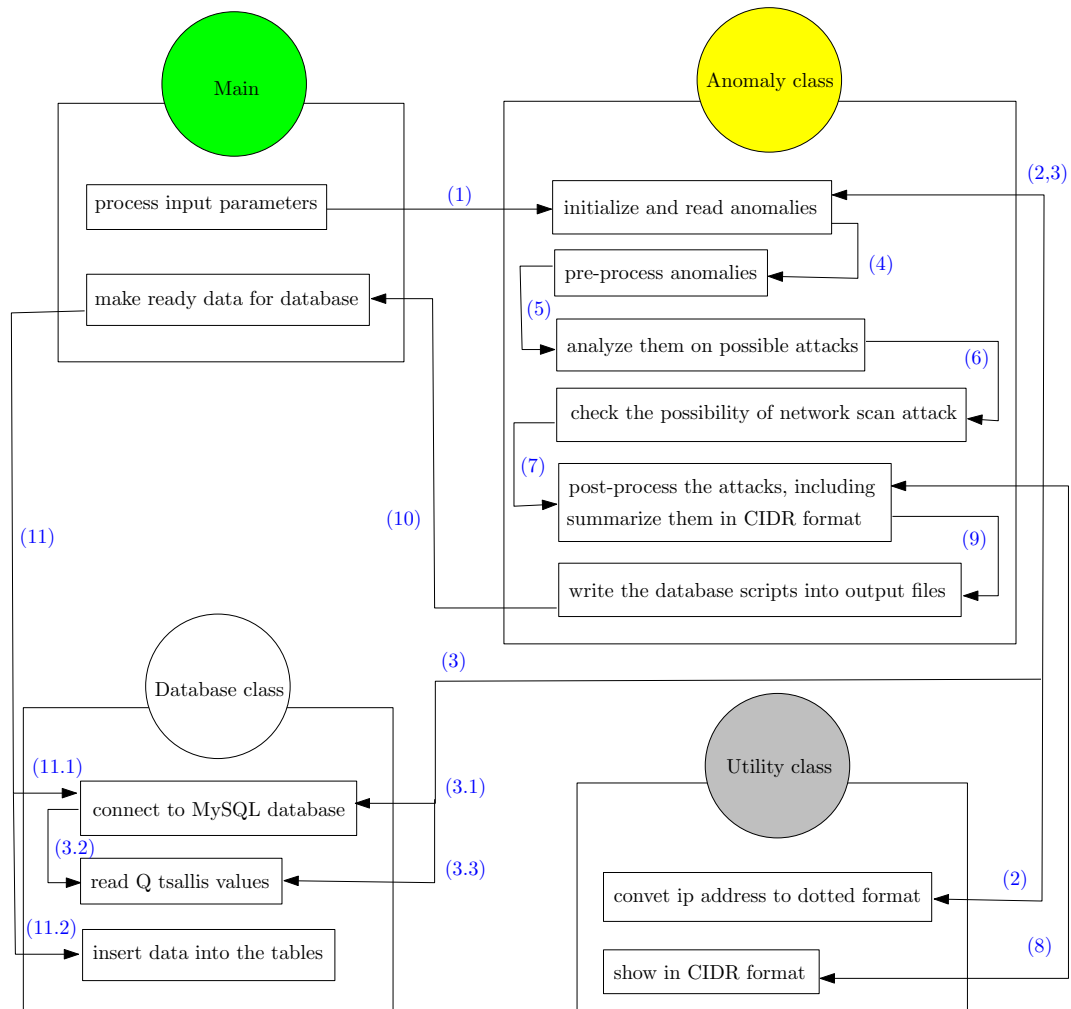


Figure 3.8: AnalyzeTES flow diagram.

```
database.o: database.cpp database.h
    $(CXX) -c database.cpp -I$(INCLUDE)
utility.o: utility.cpp utility.h
    $(CXX) -c utility.cpp -I$(INCLUDE)
clean:
    rm -rf *.o $(PROG)
```

Just executing the command *make* in the Linux shell suffices (for Windows it is similar, depends on the compiler though).

3.3.2 Usage

In order to use the AnalyzeTES program properly, these arguments should be feeded as input parameters:

```
./AnalyzeTES -f [filename] -{a,i,o,?}
These options mean:
  -a: automatically read the analysis_* files and process them
  -i: if the file contains incoming traffic
  -o: if the file contains outgoing traffic
  -?: for this guide
```

Note: *-i* and *-o* can not come together!

No need for explanations though, the program accepts an input file name (by *-f*, the output of FlowSketches program), and also either *-i* or *-o* to indicate the input file contains incoming traffic or outgoing traffic, respectively.

Chapter 4

Experimental Result

Two approaches are considered to evaluate and examine TES. First one is to take every anomaly discovered by TES visualization tools and run the FlowSketches at that particular time to see if it is really an attack or not (if not, it means false positive). Second one is to run FlowSketches and examine the q Tsallis value of every discovered attack (and also the TES pattern of the attack time period) to see whether this attack is viewable there or not (if not, it means false negative). Section 4.1 gives a successful example of the first one, and subsequently section 4.2 illustrates a unsuccessful example of the second one.

For the privacy purposes, the IP addresses are anonymized.

4.1 First approach: examining an anomaly from TES

We illustrate a sample running of the first approach, giving an example on real data. Here we would like to examine whether an anomaly seen in TES visualization tools is really an attack or not. If it is, we would like to extract the feature information about it.

We run the TES visualization program (which has been implemented in MATLAB) on the data "2008_04_12_refl_DDoS" in order to search for possible anomalies. As appears in the Figure 4.1, we can see the computed Tsallis entropy values at different point in time. Here, we concentrate on anomalies of TCP packets for incoming traffic to SWITCH network (indicated by IN tab). The TES diagram at this figure (bottom of Figure 4.1) and Figure 4.2 respectively show the entropy changes for the source IP address and the corresponding destination IP address.

Figure 4.3(a) represents a vast changes (here "strange" low activity) from the entropy of normal traffic for *source IP addresses*, and respectively Figure 4.3(b) shows a vast changes (here "strange" high activity) for *destination IP addresses*, at that particular time. Different colors show different numbers for q Tsallis values, for instance the dark blue is a value between $q = 1.5$ and $q = 1$, and the red is for values around $q = -1.75$.

For the sake of brevity, in the following, we focus on the source IP addresses. First, the start and end time of the anomaly period are extracted:

1. start time: Wednesday, April 02, 2008 10:06:46 AM (or 1207123606 in epoch format), and
2. end time: Wednesday, April 02, 2008 5:45:26 PM (or 1207151126 in epoch format).

Then, in order to run the FlowSketches program with appropriate parameters, the following script is being prepared (*START* is the start time of evaluation on the data, *INT* is time interval, *LEN* is the sketch length, and *HSH* is the number of hash functions):

```
#!/bin/sh
SKETCHDIR="/home/amahdi/fromevelyn/sketch-1.0/src"
READERDIR="/home/amahdi/work/workspace/Netflowv5Reader/Debug"
MERGERDIR="/home/amahdi/work/workspace/FlowMerger/Debug"
DATA1="/largefs1/netflow_data/raw/2008_04_12_refl_DDoS/\
19991*_1207123*.dat.bz2"
DATA2="/largefs1/netflow_data/raw/2008_04_12_refl_DDoS/\
19993*_1207123*.dat.bz2"
```

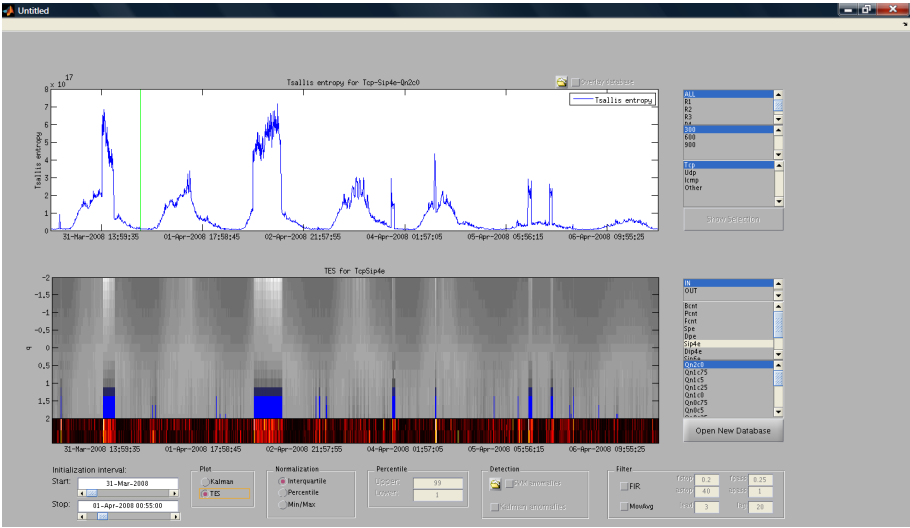


Figure 4.1: A snapshot of TES visualization program, for incoming traffic of source IP addresses.

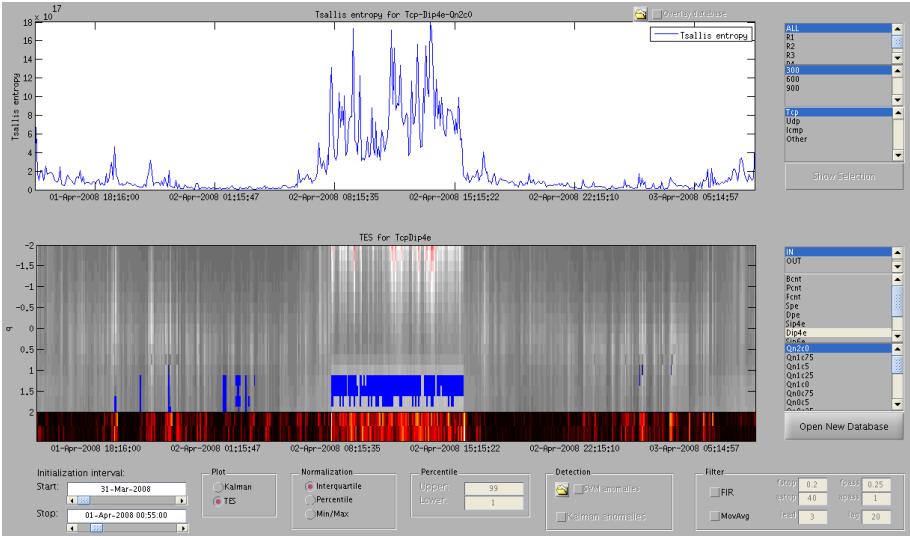


Figure 4.2: A snapshot of TES visualization program, for incoming traffic of destination IP addresses.

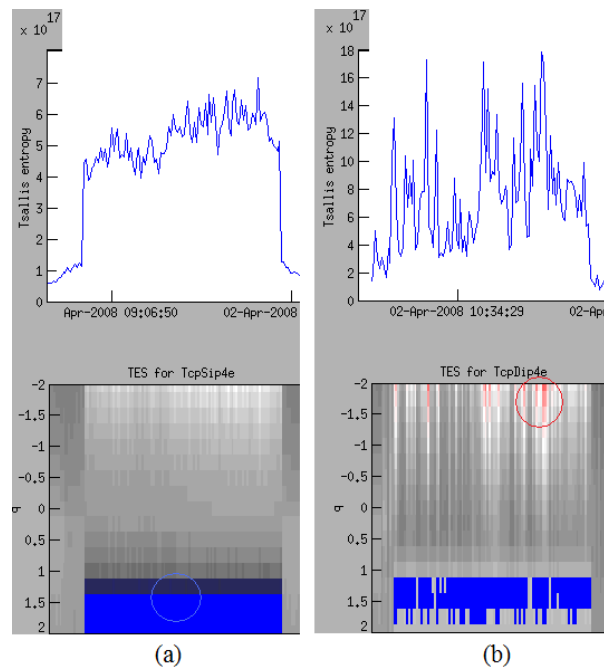


Figure 4.3: A possible anomaly discovered by TES: (a) for the source IP addresses, (b) for the destination IP addresses.

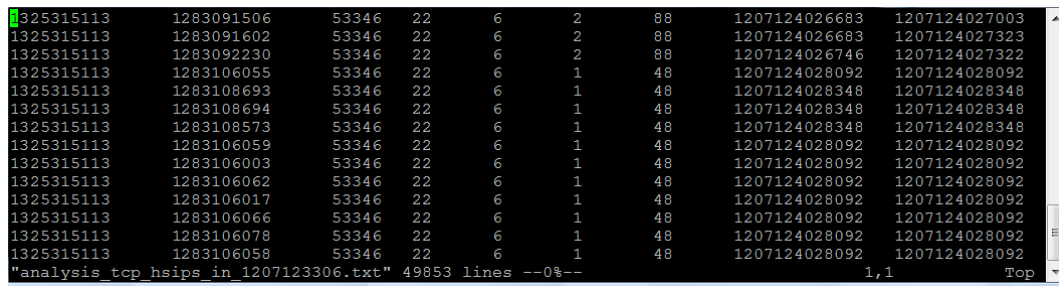
```
START=1207123606
INT=300
LEN=1024
HSH=5
# Create pipes
mkfifo pipe1
mkfifo pipe2
mkfifo pipe3
$READERDIR/Netflowv5Reader $DATA1 pipe1 &
$READERDIR/Netflowv5Reader $DATA2 pipe2 &
$SKETCHDIR/FlowSketches pipe3 /dev/null -h $HSH -l $LEN \
-i $INT -s $START &
$MERGERDIR/FlowMerger pipe3 pipe2 pipe1
# Delete pipes
rm pipe1
rm pipe2
rm pipe3
```

It is worth mentioning that FlowSketches program can use two threshold values, one considering 95%, the other considering 99% of the intervals as normal traffic. Here, we use the second threshold.

The script is run and after considerable amount of time, it writes the output in files with prefix “sketch_” (e.g. sketch_tcp_srcip_in_1.csv) for different protocols (TCP, UDP, ICMP), different features (source ip, etc), and different directions (incoming or outgoing). The collect_anomalies MATLAB program is subsequently invoked on the sketch files. After successful execution, it eventually gives out files with prefix “anomalies_” (e.g. anomalies_tcp_srcip_in_99.txt) along with text files.

At this stage, we deal with the feature extraction using the FlowExtractor program. But, before we go on with it, we need a correct Python script as the extraction scenario. For this purpose, a Perl script is changed and adapted for the current data. The result of this Perl script (named generate.pl) is two files: 1-extract.sh (as comes below), and 2-analysis1.py (the mentioned Python script).

```
#!/bin/sh
```



1325315113	1283091506	53346	22	6	2	88	1207124026683	1207124027003
1325315113	1283091602	53346	22	6	2	88	1207124026683	1207124027323
1325315113	1283092230	53346	22	6	2	88	1207124026746	1207124027322
1325315113	1283106055	53346	22	6	1	48	1207124028092	1207124028092
1325315113	1283108693	53346	22	6	1	48	1207124028348	1207124028348
1325315113	1283108694	53346	22	6	1	48	1207124028348	1207124028348
1325315113	1283108573	53346	22	6	1	48	1207124028348	1207124028348
1325315113	1283106059	53346	22	6	1	48	1207124028092	1207124028092
1325315113	1283106003	53346	22	6	1	48	1207124028092	1207124028092
1325315113	1283106062	53346	22	6	1	48	1207124028092	1207124028092
1325315113	1283106017	53346	22	6	1	48	1207124028092	1207124028092
1325315113	1283106066	53346	22	6	1	48	1207124028092	1207124028092
1325315113	1283106078	53346	22	6	1	48	1207124028092	1207124028092
1325315113	1283106058	53346	22	6	1	48	1207124028092	1207124028092

"analysis_tcp_hsipt_in_1207123306.txt" 49853 lines --0%-- 1,1 Top

Figure 4.4: A snapshot from the output result of FlowExtractor (IP addresses are anonymized).

```
# Create pipes
mkfifo pipel
# Start programs
/home/amahdi/work/workspace/Netflowv5Reader/Debug/Netflowv5Reader
/largefs1/netflow_data/raw/2008_04_12_refl_DDoS/\
  19991_00045523_1207123200.dat.bz2 \
  /largefs1/netflow_data/raw/2008_04_12_refl_DDoS/\
  19993_00045523_1207123200.dat.bz2 pipel &
/home/amahdi/work/workspace/FlowExtractor/Debug/FlowExtractor pipel \
  /dev/null analysis1.py
# Delete pipes
rm pipel
```

As mentioned in this script, FlowExtractor program is invoked on the specified data and follows the instruction given by analysis1.py Python script. The outcome is several files with prefix “analysis_” (e.g. analysis_tcp_hsipt_in_1207123306.txt) that each contains these information (the examples are from the first row of Figure 4.4):

- source IP address, e.g. 1325315113
- destination IP address, e.g. 1283091506
- source port, e.g. 53346
- destination port, e.g. 22
- protocol, e.g. 6 (for TCP)
- number of packets, e.g. 2
- size in bytes, e.g. 88
- start time, e.g. 1207124026683 (or Wednesday, April 02, 2008 10:13:46 AM)
- end time, e.g. 1207124027003 (or Wednesday, April 02, 2008 10:13:47 AM)

Figure 4.4 shows a snapshot of one of the outputs.

At this stage: In order to find out a possible explanation for the anomalies and hence insert them all into a (MySQL) database, the AnalyzeTES is invoked on the “analysis_” output files. This program has the option to insert directly the result into a database or write them down in (My)SQL language into files. In the first case, it uses MySQL C API [3] to connect, create and insert data into a MySQL database. In the second case, it generates at least these files:

1. Text file tables.txt: contains SQL commands to create the necessary tables to keep the data. The definition of main tables are mentioned in Section 3.1.
2. Text file anomalies.txt: contains SQL commands to insert anomaly data into Anomaly table (see section 3.1.1). For example:


```

INSERT INTO Anomaly VALUES ('026.065.121.200', '173.194.000.050',
    53346, 22, 6, 2, 88, 1207124026683, 1207124027003, 1.5);
INSERT INTO Anomaly VALUES ('026.065.121.200', '173.194.000.146',
    53346, 22, 6, 2, 88, 1207124026683, 1207124027323, 1.5);
INSERT INTO Anomaly VALUES ('026.065.121.200', '173.194.003.006',
    53346, 22, 6, 2, 88, 1207124026746, 1207124027322, 1.5);
INSERT INTO Anomaly VALUES ('026.065.121.200', '173.194.003.009',
    53346, 22, 6, 2, 88, 1207124026746, 1207124027386, 1.5);
INSERT INTO Anomaly VALUES ('026.065.121.200', '173.194.003.007',
    53346, 22, 6, 2, 88, 1207124026746, 1207124027322, 1.5);

```

3. Text file attacks.txt: contains SQL commands to insert the discovered/identified attacks into Attack table (see section 3.1.2).

```

INSERT INTO Attack VALUES ('173.194.000.000', '026.065.121.200',',
    ',53346,', ',22,', 6, 2, 96, 0, 1207124026683, 1207124027003,
    1.5, 'Network Scan attack,');
INSERT INTO Attack VALUES ('173.194.000.001', '026.065.121.200',',
    ',53346,', ',22,', 6, 2, 96, 0, 1207124026683, 1207124027323,
    1.5, 'Network Scan attack,');
INSERT INTO Attack VALUES ('173.194.000.002', '026.065.121.200',',
    ',53346,', ',22,', 6, 2, 96, 0, 1207124026746, 1207124027322,
    1.5, 'Network Scan attack,');
INSERT INTO Attack VALUES ('173.194.000.003', '026.065.121.200',',
    ',53346,', ',22,', 6, 2, 96, 0, 1207124026746, 1207124027386,
    1.5, 'Network Scan attack,');
INSERT INTO Attack VALUES ('173.194.000.004', '026.065.121.200',',
    ',53346,', ',22,', 6, 2, 96, 0, 1207124026746, 1207124027322
    1.5, 'Network Scan attack,');

```

4. Text file netscanAttacks.txt: contains SQL commands to insert the discovered/identified network scan attacks into NetScanAttack table (see section 3.1.3). In the following, the range of IP addresses (in CIDR notation) that the attacker '026.065.121.200' probes is listed (this list is cut after passing the specified threshold/limit). This is actually the result of running AnalyzeTES on the corresponding response traffic (indicated with “_answer.txt” among FlowSketches output files).

```

INSERT INTO NetScanAttack VALUES
('026.065.121.200', 'Randomly Distributed,
173.194.000.001/24,173.194.001.001/24,173.194.002.001/24,
173.194.003.001/24,173.194.004.001/24,173.194.005.001/24,
173.194.006.001/24,173.194.007.001/24,173.194.008.001/24,
173.194.009.001/24,173.194.010.001/24,173.194.011.001/24,
173.194.012.001/24,173.194.013.001/24,173.194.014.001/24,
173.194.015.001/24,173.194.016.001/24,173.194.017.001/24,
173.194.018.001/24,173.194.019.001/24,173.194.020.001/24,
173.194.048.001/24,173.194.049.001/24,173.194.050.001/24,...',
49152, 1207124026683, 1207124128115);

```

As indicated in the attack script example (which contains a small portion of all detected ones¹), most of them are categorized as “Network Scan attack” attacks. It means that at this time, the attacker is scanning the SWITCH network for possible victims to be probably used as a reflector for intended attack.

4.2 Second approach: example of false negative

Here, we follow a similar procedure as described for the first approach. First, FlowSketches program is run on the traffic and then the discovered attacks are evaluated and examined based

¹The tag “Randomly Distributed” is being inserted after the length of this field exceeds from a predefined threshold.

on their q Tsallis values.

So, as before, FlowSketches with appropriate parameters and data is invoked (as can be seen, we configure this script to run on all data indicated by “19991*.dat.bz2” and “19993*.dat.bz2”, not just specific ones).

```
#!/bin/sh
SKETCHDIR="/home/amahdi/fromevelyn/sketch-1.0/src"
READERDIR="/home/amahdi/work/workspace/Netflowv5Reader/Debug"
MERGERDIR="/home/amahdi/work/workspace/FlowMerger/Debug"
DATA1="/largefs1/netflow_data/raw/2008_04_12_refl_DDoS/19991*.dat.bz2"
DATA2="/largefs1/netflow_data/raw/2008_04_12_refl_DDoS/19993*.dat.bz2"
START=1206896400
INT=300
LEN=1024
HSH=5
#####
# Create pipes
mkfifo pipe1
mkfifo pipe2
mkfifo pipe3
$READERDIR/Netflowv5Reader $DATA1 pipe1 &
$READERDIR/Netflowv5Reader $DATA2 pipe2 &
$SKETCHDIR/FlowSketches pipe3 /dev/null -h $HSH -l $LEN \
-i $INT -s $START &
$MERGERDIR/FlowMerger pipe3 pipe2 pipe1
# Delete pipes
rm pipe1
rm pipe2
rm pipe3
```

After it is done with the FlowSketches, we continue with collecting anomalies as well as building the script for FlowExtractor. This script is quite similar to the above version, just with different input files. In this example we employ a smaller anomaly threshold in the collecting phase, on purpose. After FlowExtractor finishes its work, we run AnalyzeTES on the result files.

In this example, we focus on the output file for UDP protocol, the destination IP address, and for incoming traffic (i.e. file analysis_udp_hdips_in_1206919500.txt).

As the result of running AnalyzeTES, it gives out an attack list that the one below mentions the occurrence of “UDP Flooding” attack:

```
INSERT INTO Attack VALUES ('161.170.184.128','Randomly Distributed,
159.049.215.245,221.115.209.082,217.164.182.216,174.102.253.218,
161.064.186.226,231.055.241.138,167.193.108.024,066.220.164.215,
134.078.182.006,211.173.121.177,200.072.115.059,OMITTED_IP,
144.103.120.020,159.165.123.107,145','',23323,29162,10636, OMITTED_PORT,
11543,', ',12055,0,23650,', ',17, 13212, 6823754, 811072,
1206919498191, 1206920399828, 0.25, 'UDP Flooding,');
```

In above script, two values are omitted for the sake of brevity: 1-*OMITTED_IP* that contains almost 50 random source IP addresses, and 2-*OMITTED_PORT* that contains almost 7000 (look like randomly) generated source port numbers.

This attack happened in 31 March 2008, between 1:24:58 AM to 1:39:58 AM (local time). But if we have a look at the corresponding q Tsallis value at this period (which is $q = 0.25$) and also to the TES visualization program (Figure 4.5), we realize that this attack could not be observed! Although TES could not detect this actual attack, choosing an appropriate threshold in FlowSketches may bring invisible attacks into observation.

Therefore, we confront with a case that it is a UDP flooding attack but TES is not able to detect it; In other words, this example shows one of the false negatives in the TES.

Summing up, we illustrate one example of correct detection of “Network Scan Attack” by TES, and one example of weakness of TES in detecting “UDP Flooding Attack” at particular time in the data.

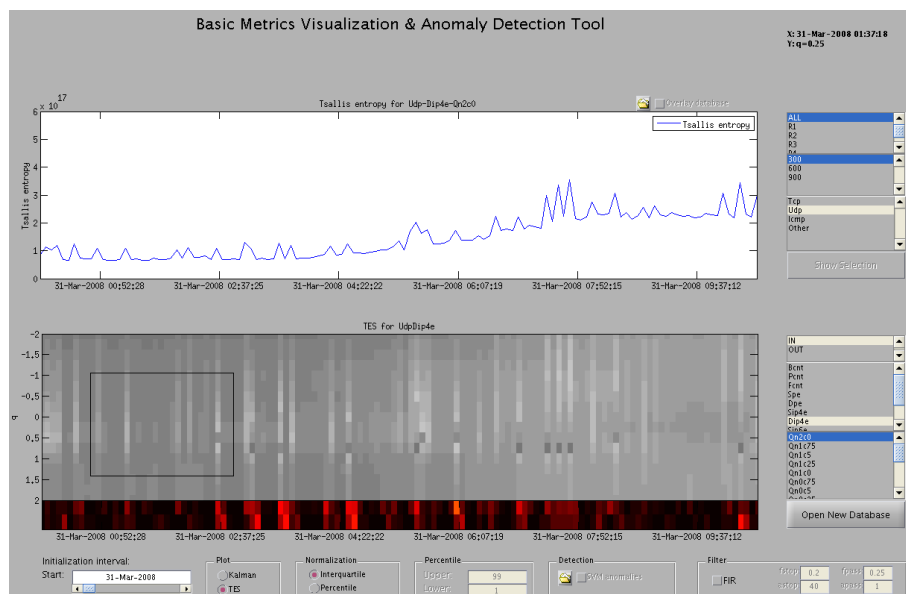


Figure 4.5: A UDP Flooding attack occurred in period that the black rectangle indicates, but it is not reflected by TES.

Chapter 5

Outlook

Some suggestions that would help to improve this work and other related works, are proposed in the following:

- The FLAME and its utility tools around are getting bigger and complicated in a bad way! As though, everybody has added something but in his own way, without following an internal standard or framework. Thus, for one single task we have to deal with several environments (and install several packages, too): running part of it with C++, with Python, some other with Perl, even with Linux shell, with MATLAB and so on. We get to do something before the whole system becomes unmanageable.
- Due to this vast spreading, this work just has been implemented in C++ and uses the existing APIs (e.g. MySQL database API) to bridge for different usages. (Besides, a program written in C++ language is so much faster in comparison to one in script languages and MATLAB.) But still major part of this work needs to use different platforms and the human involvement, which would be better to have them automated in just C++. So, the ideal version of this work can be only one program/package, providing these capabilities:
 1. *AnalyzeTES -s 1 -d database -t threshold -o pipe1*
Step one: the outcome of this running is a file (or *pipe1*) that contains all the timestamp that the *q* Tsallis value in those points does not satisfy this *threshold*. This part considers the pre-computed TES *database* files.
 2. *AnalyzeTES -s 2 -d rawdatabase -i pipe1 -o pipe2*
Step two: it takes the timestamp output of the previous step (from *pipe1*) and gives out the feature distribution for each timestamp, queried from *rawdatabase*. For this purpose, it makes use of the FlowSketches as well as FlowExtractor programs.
 3. *AnalyzeTES -s 3 -d resdatabase -i pipe2*
Step three: it takes the anomaly information generated in the previous step (from *pipe2*) and works on the data. It analyzes the file and puts a good possible explanation for every anomaly and finally inserts them into the designed database *resdatabase*.
- In the AnalyzeTES program, set of rules to identify an anomaly have been written in *if* statements (e.g. *if (tcpport == 80) then etc*). So, whenever we want to include another attack we should change the code and subsequently run the make file. A subtler idea is to define a simple language for these rules and put them into a file, and feed this file as input parameter to the program. One entry of that file could look like this: *udpsport=4000, packetsizethreshold>300 => 'X attack'*. Thus, there is no need to change and recompile the code anymore, however coming up with this rule language may not be straight-forward.
- In the existing scenario of extracting feature information, we have to read the data twice: once we run FlowSketches and the other time when we run FlowExtractor. Each of this reading and the necessary computations take considerable times in the scale of some days (it depends on the data). This way, it could not be seen as an “online” detection and identification system! It seems that we can improve the system and use a better memory management in such a way that all of the tasks are done in one reading/shot. We can

put a limit, say a window time, and only consider the data in this window to compute the threshold and the like. Along with this sort of changes, we can also employ some virtual memory management algorithms if we usually run out of memory.

- Because of time limit, we could not run this scenario on more data from different time slots. However the running may seem a bit time consuming, the scenario is almost straight-forward.

Chapter 6

Summary and Conclusion

We carried out a network anomaly analysis by the TES (Traffic Entropy Spectrum) on some traffic distributions from the SWITCH network. First, as explained mostly in Chapter 2, we took the anomalous peaks from TES and extracted their occurrence period time and their q Tsallis value. Then, we executed FlowSketches program on the specified point in times to acquire the detected anomalies by this tool as well. Subsequently, by means of FlowExtractor program, features information about the anomalous peaks gave out as output files, each of which contains IP source and destination addresses, port source and destination numbers and so on, for every protocol.

Having these feature distributions, we run the AnalyzeTES utility program to further process them and find out a reason for an anomaly or mostly for a group of anomalies. This program could help us figure out what was really going on on the traffic, and further check the TES statement in this respect. As mentioned, we were somehow able to verify either that the anomalous peak/pattern viewed in TES was really an attack, or there should be an anomalous peak/pattern in that particular time in TES but it was not. As mentioned in the (result) Chapter 4, we illustrated two examples: one (i.e. network scan attack) for the case that we could confirm the TES alarm was also verified by another tools FlowSketches, and another example (i.e. UDP flooding attack) to show how TES could be wrong about an anomaly/attack which did really occur in the system.

And finally, AnalyzeTES program stored all the findings in different output files, including the anomalies, attacks, network scan attacks, and table creation all in MySQL language. Also, it inserted them into the designed MySQL database, if the database be ready and up. By doing so, we provided a capability of further working and investigating on the anomalies and attacks in a better and efficient way of receiving assistance from database management system.

Appendix A

Implementation details

More implementation details about *AnalyzeTES* program will be given in the following.

A.1 Main

The main program, first evaluates the input arguments and if they are correct, then based on those arguments, it calls the appropriate functions.

A.1.1 run function

Function *run* has the following information:

- Definition: *int run(const char* filename, bool incoming)*
- Description: This function runs the *init* and *process* functions of Anomaly class on every input file.
- Parameters: *filename* represents the name of input anomaly file. And parameter *incoming* indicates if the traffic is coming to the inside of local network, or is going outside of it.
- Return: It returns an integer as error code.

A.1.2 autoRunning function

Function *autoRunning* has the following information:

- Definition: *int autoRunning()*
- Description: This function reads the input file names and subsequently calls the *run* function (as mentioned above) on each of them.
- Parameters: Nothing.
- Return: It returns an integer as error code.

A.2 Anomaly class

This class has the following definition. Description of each method will come at the subsequent subsections.

```
#ifndef ANOMALY_H_
#define ANOMALY_H_
#include <iostream>
#include <fstream>
#include <list>
```

```

#include <string>
#include <sstream>
#include <map>
#include <time.h>
#include <math.h>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include "utility.h"
#include "database.h"

using namespace std;

class Anomaly
{
public:
    /** data structure for keeping the processed data,
     * attackers along with the victim, and a reason/type of that attack.
     */
    struct KnownAttackNode {
        string reason;
        string victim;
        string attackers;
    };

    /** data structure for keeping the network scan attack data,
     * to put together the attacker and its target scanning victims.
     */
    struct NetScanAttackNode {
        string victims;
        int n;
        uint64_t stime;
        uint64_t etime;
    };

    /** data structure for keeping all the processed data:
     * from attackers IP addresses, the victim, total packets, and so on.
     * By processing these data, Anomaly class is able to detect a
     * specific attack by cross-checking every entry with the known
     * attack informations.
     */
    struct ProcessedInfo {
        string attackerIp;
        string attackerPort;
        string victimPort;
        uint16_t proto;
        int totalPackets;
        uint32_t totalSize;
        uint64_t duration;
        double q;
        string reason;
    };

    /** data structure for keeping the raw data read
     * from the output files.
     */
    struct Info {
        string srcip;
        string dstip;
        uint16_t srcport;
    };
};

```

```

        uint16_t dstport;
        uint16_t proto;
        uint16_t packets;
        uint16_t size;
        uint64_t stime;
        uint64_t etime;
        double q;
    };
#define DOTTED_IP_SIZE 15 //000.000.000.000
private:
    //database related
    Database db;
    string DBAnomalyTableName;
    string DBAttackTableName;
    string DBNetScanAttackTableName;
    string DBAnomalyInfoTableName;
    //network related
    string filename;
    bool IN;
    bool OUT;
    Utility util;
    string insideIP;
    string insideNetMask;
    int noIpsThreshold;
    int portScanThreshold;
    uint64_t longDurationThreshold;
    uint32_t trafficThreshold;
public:
    enum PROTO {TCP = 6, UDP = 17, ICMP = 1};
    enum ANOMALY_LIST {
        DOS,
        ICMP_FLOODING,
        ICMP_REFLECTOR,
        TCP_SYN_FLOODING,
        UDP_FLOODING,
        DNS_REFLECTOR,
    };
private:
    map<string, ProcessedInfo> attacks;
    map<string, NetScanAttackNode> netScanAttacks;
    list<Info> anomlist;
public:
    Anomaly();
    ~Anomaly();
    //read and process anomalies
    void init();
    void init(const char* inputFile, bool incoming);
    int process();
    void processAnomalies();
    void postProcessAnomalies();
    int analyze();
    //related to network scan attack
    int getNoVictimsByAttacker(string attackerIp);
    bool checkForNetScanAttack(string attackerIp);
    //print and utility functions
    bool isInside(string ipstr);
    int readAnomalies(const char *inputfile);
    void printAnomalies();

```

```

void printAttacks();
void printNetScanAttacks();
void printTimes();
void test();
//writing to a SQL script
int writeDBScriptUnprocessed(const char *outputfile);
int writeDBScriptProcessed(const char *outputfile);
int writeDBScript(const char *outputfile);
int writeDBScriptNetScan(const char *outputfile);
//requesting q Tsallis values from Database class
int setTsallisEntropy();
};
#endif /* ANOMALY_H_ */

```

A.2.1 init: first function

Function *init* has the following information:

- Definition: *void Anomaly::init()*
- Description: This function initializes the private values.
- Parameters: Nothing.
- Return: Nothing.

A.2.2 init: second function

Function *init* has the following information:

- Definition: *void Anomaly::init(const char* inputFile, bool incoming)*
- Description: This function initializes the private values.
- Parameters: *incoming* is true if the traffic is from outside to the inside of the private network. And *inputFile* contains the name of the anomaly file.
- Return: Nothing.

A.2.3 isInside function

Function *isInside* has the following information:

- Definition: *bool Anomaly::isInside(string ipstr)*
- Description: This function checks if an IP is inside of the network or outside.
- Parameters: *ipstr* string representation of an IP address in dotted format.
- Return: It returns true if it's inside, otherwise it returns false.

A.2.4 readAnomalies function

Function *readAnomalies* has the following information:

- Definition: *int Anomaly::readAnomalies(const char *inputfile)*
- Description: This function reads anomalies from the given input file and stores them in *anomlist* storage.
- Parameters: *inputfile* is an input file name.
- Return: It returns error if something bad happens during file opening.

A.2.5 printAnomalies function

Function *printAnomalies* has the following information:

- Definition: *void Anomaly::printAnomalies()*
- Description: This function prints the anomalies from *anomlist*, mostly for debugging purposes.
- Parameters: Nothing.
- Return: Nothing.

A.2.6 printTimes function

Function *printTimes* has the following information:

- Definition: *void Anomaly::printTimes()*
- Description: This function prints “time” field of *anomlist* data structure in human readable format.
- Parameters: Nothing.
- Return: Nothing.

A.2.7 processAnomalies function

Function *processAnomalies* has the following information:

- Definition: *void Anomaly::processAnomalies()*
- Description: This function processes the *anomlist* and brings together those have the same destination IP address (as the victim), by accumulating the source IP addresses, the source and destination port addresses, summing up the number of packets, calculating the amount of data being received, and measuring the duration anomaly time. Mainly, it works on *anomlist* data structure, and fills the attacks data structure as a processed anomaly list.
- Parameters: Nothing.
- Return: Nothing.

A.2.8 postProcessAnomalies function

Function *postProcessAnomalies* has the following information:

- Definition: *void Anomaly::postProcessAnomalies()*
- Description: This function post-processes the *attacks* data structure and further organizes it, such as summarizing the list of IP addresses in the CIDR format.
- Parameters: Nothing.
- Return: Nothing.

A.2.9 analyze function

Function *analyze* has the following information:

- Definition: *int Anomaly::analyze()*
- Description: This function analyzes every entry of *attacks* data structure in order to find a good enough attack description for that anomaly. The well-known attacks are listed in Tables 2.1 and 2.2.
- Parameters: Nothing.
- Return: Nothing.

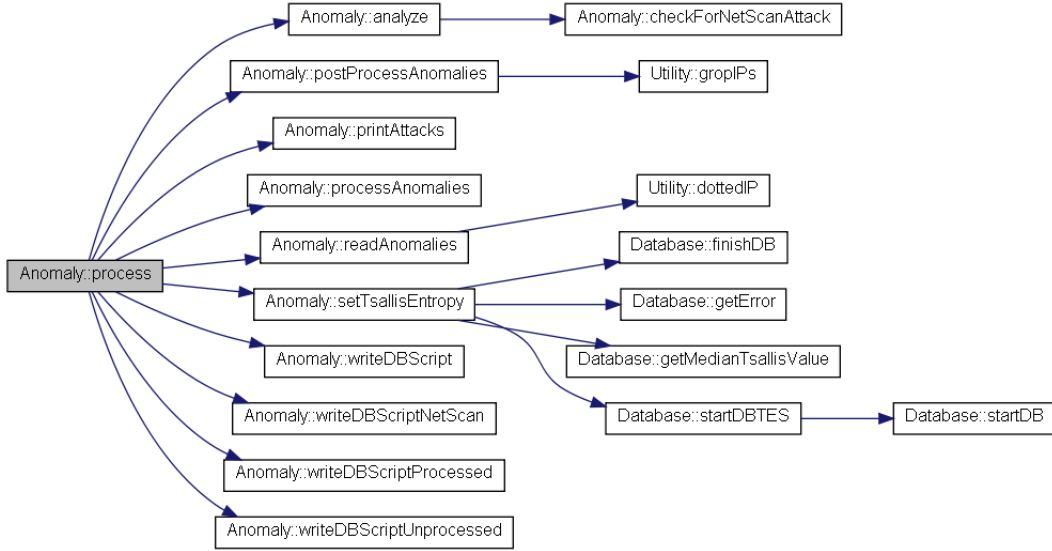


Figure A.1: Call graph of `Anomaly::process` function.

A.2.10 process function

Function *process* has the following information:

- Definition: `int Anomaly::process()`
- Description: This function is the main function in the class. It reads the input files which are the output of *FlowSketches* program, and calls *setTsallisEntropy* function in order to acquire the corresponding q Tsallis values. It subsequently executes *processAnomalies* to categorize the victims and then *analyze* to identify the attack being done in that entry. Finally, it calls database related functions in order to store the data in a database, or save the database script in a script file (to be executed on a database, later on). Along with the aforementioned functions, it often prints the intermediate step outputs for debugging purposes.
- Parameters: Nothing.
- Return: It returns an error integer. If an error occurs when it tries to open a file or the like, it returns -1, and if it ends normally, it returns 0.

Figure A.1 shows the call graph of this function.

A.2.11 printAttacks function

Function *printAttacks* has the following information:

- Definition: `void Anomaly::printAttacks()`
- Description: This function prints the entries in *attacks* data structure, in a decorated text format.
- Parameters: Nothing.
- Return: Nothing.

A.2.12 printNetScanAttacks function

Function *printNetScanAttacks* has the following information:

- Definition: `void Anomaly::printNetScanAttacks()`

- Description: This function prints the entries in *netScanAttacks* data structure, in a decorated text format.
- Parameters: Nothing.
- Return: Nothing.

A.2.13 writeDBScriptUnprocessed function

Function *writeDBScriptUnprocessed* has the following information:

- Definition: *int Anomaly::writeDBScriptUnprocessed(const char *outputfile)*
- Description: This function writes the content of *anomlist* data structure in a file, in the SQL format. It's actually a database script, so by feeding it to a database, the discovered anomalies are being stored there.
- Parameters: *outputfile* is the name of output file.
- Return: It returns an integer as error message, for the case that opening a file causes a problem.

A.2.14 writeDBScriptProcessed function

Function *writeDBScriptProcessed* has the following information:

- Definition: *int Anomaly::writeDBScriptProcessed(const char *outputfile)*
- Description: This function writes the content of *attacks* data structure in a file, in the SQL format. It's actually a database script, so by feeding it to a database, the discovered attacks are being stored in that database.
- Parameters: *outputfile* is the name of output file.
- Return: It returns an integer as error message, for the case that opening a file causes a problem.

A.2.15 writeDBScriptNetScan function

Function *writeDBScriptNetScan* has the following information:

- Definition: *int Anomaly::writeDBScriptNetScan(const char *outputfile)*
- Description: This function writes the content of *netScanAttacks* data structure in a file, in the SQL format. It's actually a database script, so by feeding it to a database, the discovered attacks are being stored in that database.
- Parameters: *outputfile* is the name of output file.
- Return: It returns an integer as error message, for the case that opening a file causes a problem.

A.2.16 writeDBScript function

Function *writeDBScript* has the following information:

- Definition: *int Anomaly::writeDBScript(const char *outputfile)*
- Description: This function writes the table definitions in a file, in the SQL format, as a database script.
- Parameters: *outputfile* is the name of output file.
- Return: It returns an integer as error message, for the case that opening a file causes a problem.

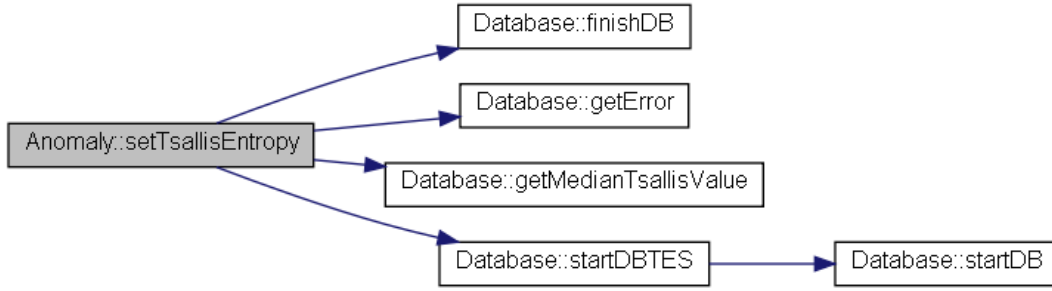


Figure A.2: Call graph of `Anomaly::setTsallisEntropy` function.

A.2.17 `setTsallisEntropy` function

Function `setTsallisEntropy` has the following information:

- Definition: `int Anomaly::setTsallisEntropy()`
- Description: This function queries the database of pre-computed Tsallis values to get the corresponding q value for the duration of that anomaly, from start time to end time. Then, adds the median value of this boundary to the `attack` data structure to be saved along with other information we have for every anomaly.
- Parameters: Nothing.
- Return: It returns an integer as error message, for the case that connection to the database raises an error. Otherwise, it returns 0.

Figure A.2 shows the call graph of this function.

A.2.18 `checkForNetScanAttack` function

Function `checkForNetScanAttack` has the following information:

- Definition: `bool Anomaly::checkForNetScanAttack(string attackerIp)`
- Description: This function checks if an attacker IP address can be marked as Network Scan attacker. If it is, the program stores them in the `netScanAttacks` data structure.
- Parameters: `attackerIp` contains the attacker IP address, in dotted format.
- Return: It returns true if this `attackerIP` be located in a network scan scenario, false otherwise.

A.2.19 `getNoVictimsByAttacker` function

Function `getNoVictimsByAttacker` has the following information:

- Definition: `int Anomaly::getNoVictimsByAttacker(string attackerIp)`
- Description: This function calculates the number of victims that are being attacked by a single attacker, given its IP address. It is mainly used for detecting the network scan attack.
- Parameters: `attackerIp` contains the attacker IP address, in dotted format.
- Return: It returns the number of victims attacked by this particular attacker.

A.3 Database class

This class has the following definition, description of each method will come at the subsequent sections.

```
#ifndef DATABASE_H_
#define DATABASE_H_
#include <iostream>
#include <string>
#include <list>
#include <sstream>
#include <fstream>
#include <vector>
#include <stdlib.h>

#define USE_DATABASE
#ifdef USE_DATABASE
    #include <mysql.h>
#endif

class Database
{
private:
#ifdef USE_DATABASE
    MYSQL *connectionDB;
    MYSQL mysql;
    MYSQL_RES *resultDB;
    MYSQL_ROW rowDB;
#endif
    list<string> dbCreateList;
    list<string> anomalyList;
    list<string> attackList;
    list<string> netScanList;
    list<string>::iterator index;
    string serverAddr;
    string userName;
    string password;
    string dbName;
    //TES related
    string TEScolumnQName;
    string TESTableName;
    string TEScolumnTimeName;
public:
    Database();
    Database(string serverAddr, string userName, string password,
            string dbName);
    ~Database();
    //initializing database handlers and so on.
    int startDB();
    int startDB(string serverAddr, string userName, string password,
            string dbName);
    int startDBTES();
    int testDB();
    void finishDB();
    string getError();
    //special function for acquiring q Tsallis value
    double getMedianTsallisValue(uint64_t startTime, uint64_t endTime);
    //read input files
    int readDBScriptAnomaly(const char *inputfile);
};
```

```

    int readDBScriptAttack(const char *inputfile);
    int readDBScriptNetScanAttack(const char *inputfile);
    int readDBScriptCreate(const char *inputfile);
    //running the script on a database
    int execDBScriptAnomaly();
    int execDBScriptAttack();
    int execDBScriptNetScanAttack();
    int execDBScriptCreate();
};
#endif /* DATABASE_H_ */

```

A.3.1 Database constructor

Constructor *Database* has the following information:

- Definition: *Database::Database(string serverAddr, string userName, string password, string dbName)*
- Description: This function is the class constructor to initialize private values.
- Parameters: *serverAddr* is the server address for connecting, *userName* is the username should be given to the connect function, *password* is the password for connecting, and *dbName* is the name of database to connect to.

A.3.2 Database destructor

Destructor *Database* has the following information:

- Definition: *Database::~Database()*
- Description: This function is the class destructor to remove the allocated memory and so on.
- Parameters: Nothing.

A.3.3 testDB function

Function *testDB* has the following information:

- Definition: *int Database::testDB()*
- Description: This function tests connection to the database and runs a simple query to check every thing is fine or not. Mostly for debugging purposes.
- Parameters: Nothing.
- Return: It returns an integer as error message, for the case that connection to the database raises an error. Otherwise, it returns 0.

A.3.4 startDB: first function

Function *startDB* has the following information:

- Definition: *int Database::startDB()*
- Description: This function makes/opens a connection to the database, and assigns a handler to the connection for hereafter use. Thus, this function is the starting point of working with database.
- Parameters: Nothing.
- Return: It returns an integer as error message: -1 for the case that connection to the database raises an error. Otherwise, it returns 0.

A.3.5 startDB: second function

Function *startDB* has the following information:

- Definition: *int Database::startDB(string serverAddr, string userName, string password, string dbName)*
- Description: This function makes/opens a connection to the database, and assigns a handle to the connection for hereafter use. Thus, this function is the starting point of working with database.
- Parameters: *serverAddr* is the server address for connecting, *userName* is the username should be given to the connect function, *password* is the password for connecting, and *dbName* is the name of database to connect to.
- Return: It returns an integer as error message: -1 for the case that connection to the database raises an error. Otherwise, it returns 0.

A.3.6 startDBTES function

Function *startDBTES* has the following information:

- Definition: *int Database::startDBTES()*
- Description: This function starts a connection to the TES database, specially to acquire q Tsallis values.
- Parameters: Nothing.
- Return: It returns an integer as error message: -1 for the case that connection to the database raises an error. Otherwise, it returns 0.

A.3.7 getError function

Function *getError* has the following information:

- Definition: *string Database::getError()*
- Description: This function returns back the last error the database class encounters in string format.
- Parameters: Nothing.
- Return: It returns the last error in string format.

A.3.8 getMedianTsallisValue function

Function *getMedianTsallisValue* has the following information:

- Definition: *double Database::getMedianTsallisValue(uint64_t startTime, uint64_t endTime)*
- Description: This function returns the median value among all q Tsallis values in the database, which are in the period of a start point to an end point.
- Parameters: *startTime* is the start time of the query, and *endTime* is the end of that query.
- Return: It returns the median value of the period.

A.3.9 readDBScriptAnomaly function

Function *readDBScriptAnomaly* has the following information:

- Definition: *int Database::readDBScriptAnomaly(const char *inputfile)*
- Description: This function reads anomalies from the given input file and stores them in *anomalyList* storage.
- Parameters: *inputfile* is the input file name.
- Return: It returns an error if something bad happens during file opening.

A.3.10 readDBScriptAttack function

Function *readDBScriptAttack* has the following information:

- Definition: *int Database::readDBScriptAttack(const char *inputfile)*
- Description: This function reads attacks from the given input file and store them in *attackList* storage.
- Parameters: *inputfile* is the input file name.
- Return: It returns an error if something bad happens during file opening.

A.3.11 readDBScriptNetScanAttack function

Function *readDBScriptNetScanAttack* has the following information:

- Definition: *int Database::readDBScriptNetScanAttack(const char *inputfile)*
- Description: This function reads network scan attacks from the given input file and store them in *attackList* storage.
- Parameters: *netScanList* is the input file name.
- Return: It returns an error if something bad happens during file opening.

A.3.12 readDBScriptCreate function

Function *readDBScriptCreate* has the following information:

- Definition: *int Database::readDBScriptCreate(const char *inputfile)*
- Description: This function reads database table description from the given input file and store them in *dbCreateList* storage.
- Parameters: *inputfile* is the input file name.
- Return: It returns an error if something bad happens during file opening.

A.3.13 finishDB function

Function *finishDB* has the following information:

- Definition: *void Database::finishDB()*
- Description: This function closes the connection already established to the database.
- Parameters: Nothing.
- Return: Nothing.

A.3.14 **execDBScriptAnomaly** function

Function *execDBScriptAnomaly* has the following information:

- Definition: *int Database::execDBScriptAnomaly()*
- Description: This function inserts the entries of *anomlist* data structure into a database. It is supposed that a connection to the database has already been established by *startDB* function (see section A.3.5).
- Parameters: Nothing.
- Return: It returns an integer as error message: -1 for the case that connection to the database is NULL or the query is not run successfully. Otherwise, it returns 0.

A.3.15 **execDBScriptAttack** function

Function *execDBScriptAttack* has the following information:

- Definition: *int Database::execDBScriptAttack()*
- Description: This function inserts the entries of attacks data structure into a database. It is supposed that a connection to the database has already been established by *startDB* function.
- Parameters: Nothing.
- Return: It returns an integer as error message: -1 for the case that connection to the database is NULL or the query is not run successfully. Otherwise, it returns 0.

A.3.16 **execDBScriptNetScanAttack** function

Function *execDBScriptNetScanAttack* has the following information:

- Definition: *int Database::execDBScriptNetScanAttack()*
- Description: This function inserts the entries of network scan attacks data structure into a database. It is supposed that a connection to the database has already been established by *startDB* function.
- Parameters: Nothing.
- Return: It returns an integer as error message: -1 for the case that connection to the database is NULL or the query is not run successfully. Otherwise, it returns 0.

A.3.17 **execDBScriptCreate** function

Function *execDBScriptCreate* has the following information:

- Definition: *int Database::execDBScriptCreate()*
- Description: This function creates tables for keeping the anomaly list and attack lists in a MySQL database. It is supposed that a connection to the database has already been established by *startDB* function.
- Parameters: Nothing.
- Return: It returns an integer as error message: -1 for the case that connection to the database is NULL or the query is not run successfully. Otherwise, it returns 0.

A.4 Utility class

This class has the following definition. Description of each method will come at the subsequent subsections.

```
#ifndef UTILITY_H_
#define UTILITY_H_
#include <iostream>
#include <string>
#include <sstream>
#include <math.h>
#include <stdlib.h>
using namespace std;
class Utility
{
public:
    #define DOTTED_IP_SIZE 15 //000.000.000.000
    string dottedIP(uint32_t ip);
    string readableTime(time_t t);
    int gropIPs(string &ips);
    string toString(const int& t);
    int fromString(const string& s);
    void test();
};
#endif /* UTILITY_H_ */
```

A.4.1 toString function

Function *toString* has the following information:

- Definition: *string Utility::toString(const int & t)*
- Description: This function converts an integer to string.
- Parameters: *t* is an input integer.
- Return: It returns string version of the input integer.

A.4.2 fromString function

Function *fromString* has the following information:

- Definition: *int Utility::fromString(const string & s)*
- Description: This function converts a string to integer.
- Parameters: *s* is a string value.
- Return: It returns integer version of the input string.

A.4.3 gropIPs function

Function *gropIPs* has the following information:

- Definition: *int Utility::gropIPs(string & ips)*
- Description: This function converts range of IPs to CIDR notation [10].
- Parameters: *ips* contains the IP addresses.
- Return: It returns an integer for error checking.

A.4.4 test function

Function *test* has the following information:

- Definition: *void Utility::test()*
- Description: This function is a test function, for debugging and nothing more.
- Parameters: Nothing.
- Return: Nothing.

A.4.5 dottedIP function

Function *dottedIP* has the following information:

- Definition: *string Utility::dottedIP(uint32_t ip)*
- Description: This function converts an integer IP to its dotted format.
- Parameters: *ip* is the integer IP address.
- Return: It returns the dotted format of IP in string representation.

A.4.6 readableTime function

Function *readableTime* has the following information:

- Definition: *string Utility::readableTime(time_t t)*
- Description: This function converts the time into a readable/user-friendly format.
- Parameters: *t* is the time in epoch format [1].
- Return: It returns the time in time-stamp format, i.e. “HH:MM:SS DD:MM:YYYY”.

Appendix B

Abbreviations

Table B.1: List of abbreviations used in text.

Abbreviation	Stands for
API	Application Programming Interface
CIDR	Classless Inter-Domain Routing
DBMS	Data Base Management System
DoS	Denial of Service
FLAME	Flow-Level Anomaly Modeling Engine
IDS	Anomaly Detection System
IP	Internet Protocol
SQL	Standard Query Language
STL	Standard Template Library
SWITCH	Swiss Academic and Research Network
TES	Traffic Entropy Spectrum

Appendix C

Timetable

Table C.1 displays the effective time table of finishing each task. To explain the tasks a little bit more: task *Theory* deals with reading the related papers [18, 20, 7, 13, 15, 9] and two master theses [14, 11] (more or less thoroughly), *FlowSketches/FlowExtrator* indicates the effort needs to correctly run these programs and get to a sound result, in *Database Design* phase a first draft of database tables were proposed and revised later, *MySQL C API* phase investigates how to query a database from a C/C++ program, *Programming* deals with developing AnalyzeTES program, *Result Collection* phase gathered necessary experimental outcome for the result part, and finally *Documentation* includes program documentation as well as preparing this report. It is worth mentioning that the tasks were not done in subsequent weeks (sometimes with one week off mostly due to technical problems).

Table C.1: Tasks and weeks used for them.

Task \ Week	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Theory	X	X													
FlowSk./FlowExt.			X	X											
Database Design					X				X						
MySQL C API						X									
Programming						X	X	X	X						
Result Collection										X	X	X			
Documentation								X				X	X	X	X

Appendix D

Task Description

D.1 Introduction

In the domain of network anomaly detection, tracking changes in feature distributions is important to many detection approaches. The problem with distributions is that they can consist of thousands of data points. This makes tracking, storing and visualizing how they change over time a difficult task. One method to cope with this problem is to use techniques to capture and describe important characteristics of distributions in a compact form. A standard technique used for this purpose is the Shannon entropy analysis. Its use for detecting network anomalies has been studied in depth and several anomaly detection approaches have applied it with considerable success. However, reducing the information about a distribution to a single number deletes important information such as the nature of the change or it might lead to overlooking a large amount of anomalies entirely. Recently, we found evidence that the Tsallis entropy offers a more detailed view on the changes in the underlying distributions. We introduce the Traffic Entropy Spectrum (TES) to analyze these changes and to use them as a basis for an anomaly detection system. However, our evaluation as well as the approach used for the detection of anomalies are incomplete since they focused on the exposure of a few large scale anomalies only. By making an in-depth analysis of a three to four week long snippet of Cisco Netflow traffic traces, this work will provide the basis for an accurate and insightful analysis of the TES.

D.2 Available data

Usually, network traces used for anomaly detection are either flow traces like e.g. specified by the Cisco NetFlow¹ [5] format or packet traces (full-payload or packet headers only). Therefore, an important precondition to do research on anomaly detection is to have such data at hand. In the course of the DDoSVax [16] project, an infrastructure to collect and store information about the network traffic crossing the borders of the Swiss Education and Research Network SWITCH [6] was set up at our institute. This dataset is a valuable source if an anomaly detection approach needs to be validated. However, because this dataset involves the information about approximately 60-200 Mio. connections per hour, it is very difficult to identify if an anomaly in a time-series or a distribution of a traffic feature is indeed an anomaly that we should care about (→ identifying ground truth!). Anomalies we should care about are mainly anomalies which involve (potentially) harmful and/or malicious activities.

D.2.1 Cluster and NetFlow Data Set

Facts about our cluster: See Figure D.1.

¹This data contains e.g. information about which Internet hosts were connected to which others and how much data was exchanged over which protocols.

Table D.1: Facts about our NetFlow data set

Coverage	March 2003 - today
Bytes/hour	500-2000 MB (compressed)
Total Bytes (02.2008)	approx. 38 TB (compressed)
Archive	Jabba (tape-library) Download speed: 5(script) to 10(backend) MB/s
Completeness	A few gaps or corrupt files (exact number unknown). An incomplete log already exists.

D.3 The Task

1. Identify anomalous peaks in a netflow trace
2. Investigation of root cause for (anomalous) peaks

The tasks (and their subtasks) are described in the following subsections.

D.3.1 Identify anomalous peaks in a netflow trace

We provide a tool (MATLAB, GUI based) to identify (statistically) anomalous peaks by looking at timeseries and distributions(spectrum) of multiple flow features (IP addresses, port numbers, flow-, packet- and byte counts). Propose a structure for storing at least the following information in a mysql DB: time, metric(s) and root cause(s) of each peak/anomaly. Note that the root cause(s) will be inserted after having completed the next step.

Expected Output

Expected output are DB design and implementation, filled with times and metric(s) of the peaks/anomalies.

D.3.2 Investigation of root cause for (anomalous) peaks

Run a detailed analysis at the identified points in time. To do this, existing tool(s) written in C++ have to be extended so that they provide information about if and how a distribution changed at these points in time (e.g., which RELEVANT IP ranges or ports were responsible for this change).

Expected Output

Expected output are extended tool and description of identified root causes in DB.

D.3.3 Optional: Improvement and tuning of the anomaly detection approach

Analyze the weaknesses recognized in the previous tasks and find ways to mitigate them. Most probably, our simple detection approach will be the most prominent weakness and is therefore likely to be the best candidate for further improvements.

D.4 Deliverables

This work has the following deliverables:

1. A concise and detailed log of the conducted work.

2. A concise and detailed documentation of all experiments and their results so that they are easy to reproduce.
3. The code and - if appropriate - installation instructions for the developed tools.
4. A ready-to-use installation on our computing cluster.

D.4.1 Documentation structure

The report should contain the steps conducted (methodology) along with findings/consequences/results, lessons learned, summary and an outlook on future work. All design decisions should be motivated and described. Any selection of parameters and their value/range should be justified. Evaluations have to be academically sound.

The code of all of the in this thesis developed tools should follow a coherent and clean coding and commenting style so that the code is easy to understand and use. If applicable, the documentation could be generated using automated documentation tools (e.g. doxygen).

D.4.2 Presentations

A final presentation at TIK will be scheduled close to the completion date of the work.

D.5 General Information

Dates/Meetings:

- This work starts on Thursday, 05.03.2009 and finishes on Monday, 29.05.2009. A total of at least 150 hours have to be spent.
- Informal meetings with the tutors will be held at least once a week.

Figure D.1 represents the network topology for the working cluster.

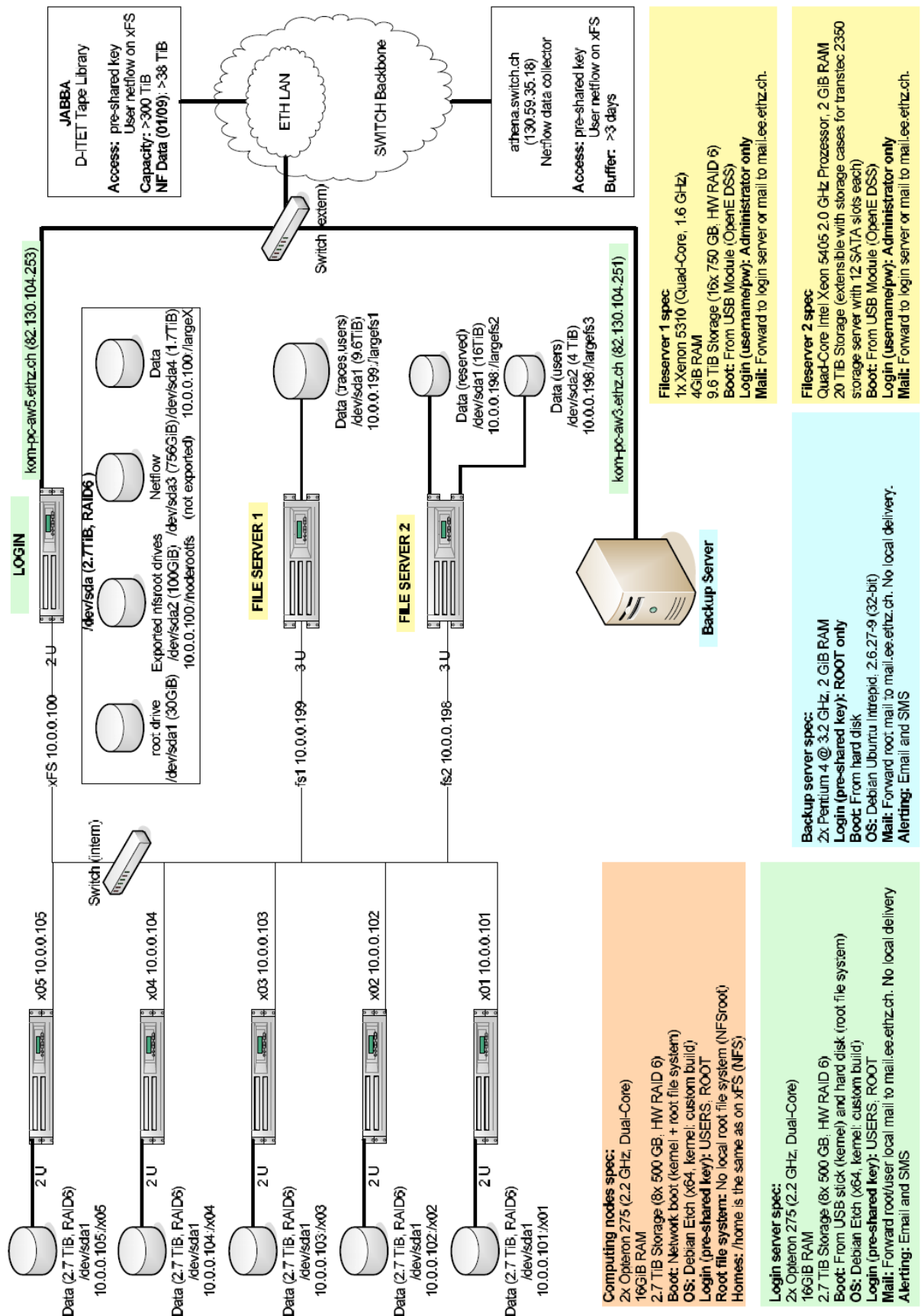


Figure D.1: Working cluster structure at TIK

Bibliography

- [1] Epoch Converter. <http://www.epochconverter.com>.
- [2] Kullback–Leibler divergence. http://en.wikipedia.org/wiki/Kullback-Leibler_divergence.
- [3] MySQL C Application Programming Interface. <http://dev.mysql.com/doc/refman/5.0/en/c.html>.
- [4] MySQL Open Source Database. <http://www.mysql.com>.
- [5] Netflow services solutions guide. <http://www.cisco.com/univercd/cc/td/doc>.
- [6] The switch education and research network. <http://www.switch.ch>.
- [7] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, page 164. ACM, 2006.
- [8] D. Brauckhoff, A. Wagner, and M. May. FLAME: A flow-level anomaly modeling engine. In *Proceedings of the conference on Cyber security experimentation and test*, page 1. USENIX Association, 2008.
- [9] X. Dimitropoulos, M. Stoecklin, P. Hurley, and A. Kind. The eternal sunshine of the sketch data structure. *Computer Networks*, 52(17):3248–3257, 2008.
- [10] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. Technical report, BCP 122, RFC 4632, August 2006.
- [11] T. Gsell. Evaluating and Improving TES. Master's thesis, Computer Engineering and Network Laboratory, ETH Zurich University, 2009.
- [12] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 219–230. ACM New York, NY, USA, 2004.
- [13] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, page 228. ACM, 2005.
- [14] E. Leuenberger. Extending FLAME: Anomaly Characterization and Extraction. Master's thesis, Computer Engineering and Network Laboratory, ETH Zurich University, 2009.
- [15] X. Li, F. Bian, M. Crovella, C. Diot, R. Govindan, G. Iannaccone, and A. Lakhina. Detection and identification of network anomalies using sketch subspaces. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, page 152. ACM, 2006.
- [16] C. Schlegel and T. Dubendorfer. UPFrame-A generic open source UDP processing framework. <http://www.tik.ee.ethz.ch/~ddosvax/upframe/>.
- [17] C.E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.
- [18] B. Tellenbach, M. Burkhart, D. Sornette, and T. Maillart. Beyond Shannon: Characterizing Internet Traffic with Generalized Entropy Metrics. In *Passive and Active Measurement Conference (PAM)*. Springer, 2009.

- [19] C. Tsallis. Possible generalization of Boltzmann-Gibbs statistics. *Journal of statistical physics*, 52(1):479–487, 1988.
- [20] A. Ziviani, ATA Gomes, ML Monsores, and PSS Rodrigues. Network anomaly detection using nonextensive entropy. *IEEE Communications Letters*, 11(12):1034–1036, 2007.