

A FRAMEWORK FOR SCORING AND TAGGING NETFLOW DATA

Submitted in partial fulfilment
of the requirements of the degree of

MASTER OF SCIENCE

of Rhodes University

Michael John Sweeney

Grahamstown, South Africa

December 2017

Abstract

With the increase in link speeds and the growth of the Internet, the volume of NetFlow data generated has increased significantly over time and processing these volumes has become a challenge, more specifically a Big Data challenge. With the advent of technologies and architectures designed to handle Big Data volumes, researchers have investigated their application to the processing of NetFlow data. This work builds on prior work wherein a scoring methodology was proposed for identifying anomalies in NetFlow by proposing and implementing a system that allows for automatic, real-time scoring through the adoption of Big Data stream processing architectures.

The first part of the research looks at the means of event detection using the scoring approach and implementing as a number of individual, standalone components, each responsible for detecting and scoring a single type of flow trait. The second part is the implementation of these scoring components in a framework, named Themis¹, capable of handling high volumes of data with low latency processing times. This was tackled using tools, technologies and architectural elements from the world of Big Data stream processing. The performance of the framework on the stream processing architecture was shown to demonstrate good flow throughput at low processing latencies on a single low end host. The successful demonstration of the framework on a single host opens the way to leverage the scaling capabilities afforded by the architectures and technologies used. This gives weight to the possibility of using this framework for real time threat detection using NetFlow data from larger networked environments.

¹ The Greek goddess of justice.

“Ay, wicked men never elude pure Themis: night and day her eyes are on them...”

- Quintus Smyrnaeus, Fall of Troy

Acknowledgements

First and foremost I would like to acknowledge and thank my partner Fiona Davern for her patience and support throughout the last year. Without her encouragement and help this would have been significantly more challenging.

Thanks must definitely also go to my supervisor, Barry Irwin, for the support and guidance he provided throughout the process. His feedback and insight has played a major part in shaping this project into its final state.

Contents

List of Figures	x
List of Tables	xi
Listings and Algorithms	xii
1 Introduction	1
1.1 Problem Statement	2
1.2 Research Objective and Goals	4
1.3 Scope	5
1.4 Document Structure	6
2 Research in Context	8
2.1 Concepts	8
2.1.1 NetFlow	8
2.1.2 Big Data and NetFlow Analysis	11
2.1.3 Stream Processing	14
2.1.4 Flow Scoring	18
2.2 Literature Survey	18
2.2.1 Using NetFlow for Security Incident Detection	19
2.2.2 Big Data and Network Security	21
2.2.3 Scoring for Network Security	24
2.3 Similar Implementations	26
2.4 Summary	27

3	Themis Design, Architecture and Requirements	28
3.1	Architecture	28
3.1.1	Macro-Architecture	29
3.1.2	Micro-Architecture	30
3.2	Themis Requirements	31
3.2.1	Data Collection	31
3.2.2	Data Ingestion	31
3.2.3	Flow Enrichment	32
3.2.4	Flow Scoring and Tagging	33
3.2.5	Utility Components	36
3.2.6	Output Components	36
3.2.7	Configuration	37
3.2.8	Information Sources	37
3.2.9	Operational Data Store	38
3.2.10	Analysis and Reporting	38
3.3	Summary	38
4	Implementation	40
4.1	Technology and Tool Discussion	40
4.1.1	Queuing Frameworks	41
4.1.2	Event Stream Processing Systems	42
4.1.3	Operational Data Store	48
4.1.4	Persistence	48
4.1.5	Languages	48
4.2	Implementation Overview	48
4.2.1	Environmental Context	49
4.2.2	Flow Scoring Process	50
4.2.3	Flow Processing Overview	51
4.3	Flow Ingestion	54

4.4	Data Sources	56
4.4.1	Internal Network Configuration	57
4.4.2	IP Address to ASN lookup	57
4.4.3	IP GeoLocation	57
4.4.4	IP Blacklists/Whitelists	58
4.4.5	Threat Intelligence Feeds	58
4.4.6	Country Watch List	59
4.4.7	TCP/UDP Services	59
4.5	Flow Enrichment Bolts	59
4.5.1	JSONtoTupleBolt	60
4.5.2	ASNBolt	60
4.5.3	GeoLocatorBolt	61
4.6	Flow Scoring	61
4.6.1	ScoreBolt	61
4.6.2	ScoreCountryBolt	62
4.6.3	ScoreDarkIPBolt	63
4.6.4	ScoreGenericIPListBolt	63
4.6.5	ScoreHTTPBruteForceBolt and ScoreSSHBruteForceBolt	64
4.6.6	ScoreInsecurePortConversationBolt and ScoreUnknownPortConversationBolt	65
4.6.7	ScoreServiceBolt	66
4.6.8	ScorePossibleScanBolt	67
4.6.9	ScoreIntelMQBolt	68
4.7	Flow Output Bolts	68
4.7.1	LoggerBolt	68
4.7.2	PersistScoredBolt	68
4.7.3	PersistScanCandidateBolt	69
4.7.4	PersistScoredBoltToKafka	71
4.7.5	PersistUnscoredBolt	71

4.8	Flow Utility Bolts	72
4.8.1	Kafka Spout (ingestion)	72
4.8.2	PersistTimerBolt	72
4.8.3	MatchFlowBolt	73
4.8.4	FlowSplitterBolt	73
4.8.5	JoinBolt	74
4.9	Flow Scoring Topology	74
4.10	Scored Flows Analysis	75
4.10.1	Visualisation	76
4.10.2	Static Analysis	76
4.11	Summary	76
5	Results and Discussion	77
5.1	Data Processing	77
5.1.1	Sample Data	77
5.1.2	Timestamp Accuracy	78
5.1.3	Topology Scoring Configuration	78
5.1.4	Processing Environment	81
5.1.5	Apache Storm Configuration	82
5.1.6	Storm Performance	83
5.2	Processing Run	86
5.3	Scoring and Tagging Results	89
5.3.1	The Data Funnel	89
5.3.2	Data Enrichment	91
5.3.3	Traffic Statistics	91
5.3.4	Score Statistics	98
5.3.5	Tag Statistics	101
5.4	Analysis of Results	105
5.4.1	Increase in Bad UDP Traffic	105

5.4.2	TCP Traffic Spike - 24th of January	106
5.4.3	Spike in Activity - 23rd to 25th of February	107
5.4.4	Top Scoring Conversations	108
5.4.5	Top Tagged Conversations	108
5.4.6	Bolt Efficacy	109
5.5	Summary	111
6	Conclusion	112
6.1	Summary	112
6.2	Research Evaluation	113
6.3	Significance of Research	114
6.4	Future Work	115
6.4.1	Enhancing the Scoring	115
6.4.2	Candidate Scan Processing	116
6.4.3	Performance Improvements	116
6.4.4	Scaling and Elasticity	117
6.4.5	Other Streaming Technologies	117
6.4.6	Analysis of Output	117
6.4.7	Production Testing	118
6.4.8	Machine Learning Opportunities	118
6.4.9	Network Traffic Accounting and Scoring	118
6.4.10	Other Uses	118
	References	119
	Appendix A Flow DDL	127
	Appendix B Input JSON Schema	128
	Appendix C Java Flow Tuple Objects	130
	Appendix D Output JSON Schemas	132

Appendix E	Logger Bolt Output	136
Appendix F	Bolts, Tags and Scores	137
Appendix G	Scoring Topology	138
Appendix H	Source Code	139

List of Figures

2.1	NetFlow tracking and counters	9
2.2	NetFlow deployment components	11
2.3	Core NetFlow attributes	12
2.4	Event time domain mapping	12
2.5	Message processing options	15
2.6	Stream processing DAG	16
3.1	Themis conceptual architecture	29
3.2	Themis high-level architecture	29
3.3	Stream processing architecture	30
3.4	Flow scores and tags	35
4.1	Themis technology stack	41
4.2	Stream of tuples	44
4.3	Tuple spout	45
4.4	Tuple bolt	45
4.5	Topology of bolts and spouts	46
4.6	Topology scaling concepts	46

4.7	Storm management UI	47
4.8	Inside and outside Network	49
4.9	Traffic categories	49
4.10	Bolt scoring	51
4.11	Flow ingestion	52
4.12	Flow scoring workflow	53
4.13	Additive scoring example	54
4.14	Scoring from external data sources	55
4.15	Output of scored flows	56
4.16	Scored flow ERD	69
4.17	Vertical scan detection	70
4.18	Horizontal scan detection	70
4.19	Botnet scan detection	71
5.1	One executor, one task per bolt	84
5.2	Bolt comparisons (1,000 tuples outstanding)	84
5.3	Bolt comparisons (40,000 tuples outstanding)	85
5.4	Four executors, four tasks per bolt. Eight for MatchFlowBolt	86
5.5	OS CPU Counters	87
5.6	Processing throughput	87
5.7	Processing latency	88
5.8	Bolt performance	89
5.9	Data reduction	90

5.10	Bad conversations as percentage of total	90
5.11	Traffic utilisation comparison	93
5.12	Bad sample as percent of total	93
5.13	Bad UDP sent traffic as percent of total	94
5.14	Bad TCP sent traffic as percent of total	94
5.15	Bad sample score distribution	99
5.16	Bad sample score distribution - expanded tail	100
5.17	Total score by day	100
5.18	Average score by day	101
5.19	Tag count per day	101
5.20	Tag count per day broken down by tag	102
5.21	Tag count per day broken down by tag (scans excluded)	102
5.22	Tag category breakdown	103
5.23	Tag count per conversation distribution	104
5.24	Tag combination counts (4 tags or more)	104
5.25	Top tagged conversation permutations	105
G.1	Flow scoring topology	138

List of Tables

1.1	Worst case flow projections	3
1.2	Flow projections with average packet size	4
2.1	NetFlow attributes	10
3.1	Minimum flow attributes	32
4.1	Scoring bolts	75
5.1	Port scoring tags	80
5.2	Service scoring tags	80
5.3	Processing Environment	82
5.4	Throughput statistics	88
5.5	Scoring categories	90
5.6	Top bad traffic sources and destinations by inside host	95
5.7	Top bad traffic sources and destinations by outside host	96
5.8	Top bad traffic sources and destinations by inside port	96
5.9	Top bad traffic sources and destinations by outside port	97
5.10	Scored flow statistics	99
F.1	Scoring bolts detail	137

Listings and Algorithms

1	Scoring and Tagging Example	62
2	Country Scoring Bolt Logic	62
3	Dark IP Scoring Bolt Logic	63
4	Generic IP List Scoring Bolt Logic	63
5	HTTP Brute Forcing	64
6	SSH Brute Forcing	65
7	Insecure Port Check	66
8	Unknown Port Check	66
9	Internally or Externally Hosted Services	67
5.1	Flow record input	91
5.2	Enriched conversation record	92
A.1	Flow DDL	127
B.1	Flow JSON	128
B.2	Flow JSON	129
C.1	Flow Tuple	130
C.2	Flow Score Class	131
D.1	Scored Flow JSON	132
D.2	Scored Flow Detail JSON	133
E.1	Log Output	136

Chapter 1

Introduction

Along with the growth of the Internet, there has been a corresponding growth in the amount of data generated (Cisco Systems, 2015b). And with this data, a corresponding increase in the amount of security related data and the associated problems with processing it. As far back as 1994 studies into data aspects of intrusion detection noted that a typical user generates as much as 35 Megabytes of data in an eight-hour period and one hour's worth of data can itself take several hours to analyse (Zuech *et al.*, 2015). The recommendations at the time were that filtering, clustering and feature selection on the data was critical in order to achieve real-time detection of incidents. Twenty three years ago security information was already being identified as a Big Data problem. Since then a lot has changed. Hardware has become cheaper, RAM and disk space have increased in size and CPU power has increased significantly, but so too has the amount of data generated by users. The volume of data has raised significant challenges for administrators in terms of how to identify threats in amongst the large volumes of network traffic, a large part of which is often background noise. The challenge is that it is easy to find the events you know about, but what about the events you don't know about yet suspect are there? The known unknowns (with apologies to Donald Rumsfeld¹). It is the known unknowns that are of greatest concern to administrators and the hardest to find.

In Sweeney and Irwin (2017) the authors propose a solution to this through a scoring and tagging approach to analysing NetFlow data. Instead of a binary approach to threat identification that requires significant pre-knowledge and classifies traffic as a threat or non-threat, a scaled view of threats is proposed where issues are viewed on a continuum

¹<http://archive.defense.gov/Transcripts/Transcript.aspx?TranscriptID=2636>

and not in isolation. A series of tests can be applied to NetFlow data, each of which by itself may not indicate serious problems. However, as some flows are scored higher and higher, the serious issues emerge. The application of many different tests also means that issues can be viewed in context. For example, network traffic on unknown ports may not be considered an issue, but when this traffic also includes high volumes of traffic from sites with low reputational scores, this may be an indication of something more serious. By applying both positive (for known or benign traffic) and negative (for suspicious traffic) scoring, the approach allows for the filtering out of normal traffic while bringing suspect traffic to the fore.

In this research, the flow-scoring approach is implemented using Big Data technology with the aim of testing the feasibility of real-time NetFlow scoring. By adopting Big Data tools and analysis techniques in combination with the scoring approach, we can advance actionable security intelligence through a reduction in the time needed for correlating, consolidating and contextualising diverse security event information.

1.1 Problem Statement

The use of NetFlow analysis for incident detection is well established (see chapter 2 for examples of prior work), however, as volumes have grown, the real-time analysis of flow data has become a significant challenge. In order to implement a real-time processing system, one first needs to establish what is meant by real-time (or even near real-time) processing. For security incident detection, this would be a function of the delay between when an event took place and when it was detected. In this work, this would be the time taken to score and tag a flow record. The range of time within which this delay falls is an indicator of how “real time” the system is. There are many different definitions for this range, with values ranging from a few milliseconds (Malaska, 2015), a few seconds (Wilson, 2015) and even minutes (Walker, 2015). In these works, the authors also make reference to near real time, but this too has a number of different definitions depending on where you look (e.g. a few hundred milliseconds or a few minutes). Any solution aiming to achieve real-time processing status would be safe to aim for processing latencies on the lower end of these values, i.e. processing time in the order of a few seconds per flow record.

Table 1.1: Worst case flow projections

Flow projections - worst case				
	1Mbps	100 Mbps	1 Gbps	10 Gbps
Bytes/second	125,000	12,500,000	125,000,000	1,250,000,000
Packets/second	3,125	312,500	3,125,000	31,250,000
Conversations/second	446	44,643	446,429	4,464,286
Flows/second	893	89,286	892,857	8,928,571

Low latencies alone in a system do not necessarily equate to real-time processing. The throughput capabilities of a system will affect the processing latency and therefore need to be considered as part of the performance measures with low latency, high throughput being the goal. In order to specify throughput requirements, we need to consider how many flows per second the framework must handle. This value can be determined empirically or theoretically. Empirical evidence is difficult to find due to the varying nature of traffic loads and so a theoretical value is simpler to calculate.

The amount of flow data generated by a network is dependent on many variables including size and usage profiles. While the speed of a network link contributes towards the number of flows generated, the nature of the traffic can have more influence on this number. Consider that a single, large download on a high-speed network will generate one flow record, while DNS lookups during log processing may generate thousands of records. In order to understand the number of records that are being generated and what sort of flow throughput could be expected, we can make some simple assumptions about traffic profiles and then estimate flows per second on different link speeds. This gives some idea of the throughputs we can expect.

Table 1.1 shows the projected flows per second that could be expected in one worst case scenario. In this case, we assume the following: a minimum of 40 bytes per packet, each conversation is considered to be a minimum of seven packets (this is a minimum for a TCP/IP conversation²) and the link is running at 100% capacity. These values are obviously not likely in the real world, as the traffic mix would be much more heterogeneous, but this allows us to estimate a general value for flows per second on different link speeds. As can be seen on a 100 Mbps link, an estimated 89,000 flows per second could be expected and this scales up to almost 900,000 on a 1 Gbps link.

²This would consist of SYN, SYN-ACK, ACK, a data packet, FIN, FIN-ACK and an ACK

Table 1.2: Flow projections with average packet size

Flow projections - average packet size				
	1Mbps	100 Mbps	1 Gbps	10 Gbps
Bytes/second	125,000	12,500,000	125,000,000	1,250,000,000
Packets/second	368	36,765	367,647	3,676,471
Conversations/second	53	5,252	52,521	525,210
Flows/second	105	10,504	105,042	1,050,420
Flows/second at 50% utilisation	53	5,252	52,521	525,210

However, this traffic profile is not realistic. Modelling for a real traffic mix is difficult and a whole area of research in itself. Research has also noted that this changes over time as applications on the Internet change (Sinha *et al.*, 2007). To better estimate the number of expected flows per second we can make some simple changes to the model in the table by adjusting three values: the number of bytes per packet, the number of packets per conversation and the average load on the link. Testing the various permutations is beyond the scope of this document. One statistic that is fairly well researched is the average packet size, which has been found to have varying characteristics depending on time, capture location and user profile (Murray and Koziniec, 2012; Mikians *et al.*, 2012). Vendors have also considered this problem and have introduced standard traffic profiles for testing load on network devices (Agilent Technologies, 2001). For our purposes we have taken a number from the lower bound of the various suggestions and proposed a new model for estimating flows per second. These values are 340 bytes per packet (taken from Agilent Technologies (2001)). If we also assume a link that is 50% congested, we come to the numbers in table 1.2. In this case, we can see that we should expect 5,000 flows per second on a 100 Mbps link and 52,000 flows per second on a 1 Gbps link. These numbers provide us with some idea of what sort of throughput our framework could be expected to deal with based on different link sizes.

1.2 Research Objective and Goals

The project has two primary goals aimed at addressing the challenge of real-time security event detection. The first goal of this project is to build upon the prototype NetFlow scoring and tagging work described in Sweeney and Irwin (2017). This will provide the means by which events will be detected and raised up for attention. In the original work, the scoring and tagging was done in a batch processing manner. Scoring and tagging of the sample data was done across large samples of data, one test at a time, and the final

output slowly built up incrementally. This is not suitable for real-world use and leads on to the second goal.

The second goal of the project is the implementation of a framework (called Themis) that enables the near real-time scoring and tagging of a stream of NetFlow data. The key driver for this requirement is the concept of perishable insights (Callaghan, 2015) - for some types of data there is only a limited amount of time to act on the information gleaned from analysis. From an information security incident detection perspective, insights derived during data processing and analytics have a limited shelf life. The longer the time gap between occurrence and detection, the greater the potential for damage and loss. Traditional approaches to data analysis focus on batch processing large sets of data periodically. This approach has two major limitations: firstly, there is the time lag between collecting the data and analysing it; and secondly, the processing of large batches of data often requires significant computational resources. In this project, the focus will be on the design and implementation of a framework that addresses these concerns providing insights as close to the time of occurrence as possible in a low-latency scalable manner. For this, we will be taking guidance from the tools and architectures employed in the world of event stream processing. The two key requirements of this framework will be its ability to handle high volumes of NetFlow data with a low end-to-end latency, while also having the capacity to scale up as volumes increase. The values for flow volume and processing latencies from the previous section can give guidance when measuring the success of this objective.

1.3 Scope

The project scope is limited to the analysis of a single data set spanning two and half months worth of NetFlow data from a small pre-existing network. The collection and pre-processing of the NetFlow data was excluded from this work and it was assumed that it could be collected and delivered to the framework in near real time using a variety of technologies. There was no foreknowledge of any incidents in this sample prior to starting the work. The enrichment and scoring components implemented have been chosen for proof-of-concept purposes in order to demonstrate a wide variety of options. Scoring values have been chosen based on broad assumptions about the implications of the anomalies detected. Only a subset (approximately 20%) of the final scored flow output was reviewed

in any detail due to time constraints. A number of significant anomalies from this sample were selected for a more detailed analysis.

During implementation technologies were chosen for their ability to scale to many hosts (horizontal scaling), however, during testing Themis was only run on a single, quad core host. Experiments with scaling were done on this host, but scaling beyond a single node was left for future work. Load testing of the framework was achieved through ingesting the entire two-and-a-half-month sample of data as fast as possible. Finally, outputs were analysed manually using a combination of charting tools and SQL queries.

1.4 Document Structure

The chapters in this document have been structured as follows:

- **Chapter 2:** This chapter consists of two parts, the first being a technology discussion and the second a literature survey. The technology section starts with a brief overview of NetFlow and then moves on to discuss its Big Data and event stream characteristics. This part concludes with a discussion of event stream processing concepts. In the second half of the chapter, a literature survey is presented. This covers the use of NetFlow in information security, Big Data and NetFlow, and scoring approaches to information security.
- **Chapter 3:** This chapter describes the requirements, design and proposed architecture of Themis. The chapter also lists the various components required in order to ingest, enrich, score and then output the flow data in order to meet the goals of the research.
- **Chapter 4:** The implementation of the design specified in chapter 3 is documented in this chapter. Details are given of the technology platforms used and how the scoring process is implemented therein. The individual elements that were implemented in order to carry out enrichment, scoring and utility components are described in detail, giving insight into the journey that flow records take through the processing framework.
- **Chapter 5:** Results and findings are presented in this chapter. The results cover the flow processing throughput and latency statistics, the operating environment

performance and then the details of the scoring outputs. As this work is a proof of concept, a select sample of the outputs of interest are analysed in detail in order to demonstrate the effectiveness of the solution and provide a view on how well the goals have been achieved.

- **Chapter 6:** The final chapter summarises the findings and results with respect to the project objectives. Opportunities for further work or extensions to the framework are also presented.

Footnotes have been used extensively throughout to provide references to the websites of software or data sources referred to in this work.

Chapter 2

Research in Context

This chapter provides the context within which this research has been carried out. In the first section a discussion of various technical concepts is presented covering NetFlow concepts, the suitability of big data approaches to NetFlow processing, stream processing and scoring for anomaly detection. In section 2 prior work is reviewed looking at the different approaches taken when analysing NetFlow data for incident detection. In section 3 a number of similar implementations are presented for comparison purposes and finally, the chapter is summarised in the last section.

2.1 Concepts

The design and architecture (as described in chapter 3) needs to be guided by the characteristics of the data processing we are undertaking and in turn, by the characteristics of the data itself. In this section, we review the structure and nature of our input data and leading on from this, consider the design and architectural elements that may best suit our implementation.

2.1.1 NetFlow

NetFlow was originally developed at Cisco in 1996 as a packet switching technology (Kerr and Bruins, 2001). In time, the company engineers realised that the data used for switching purposes could also be leveraged to provide detailed network traffic utilisation data.

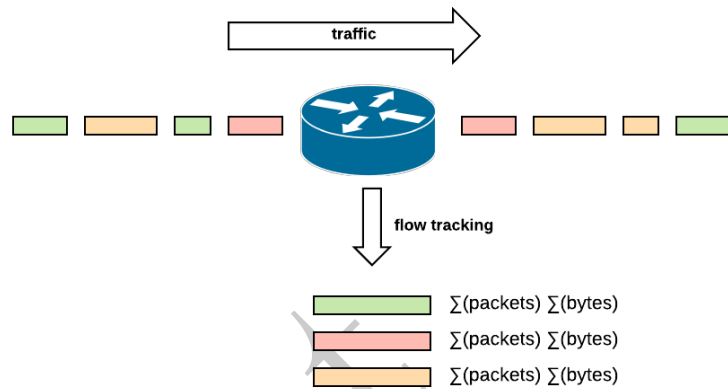


Figure 2.1: NetFlow tracking and counters

As illustrated in figure 2.1, packet counters used for fast switching purposes were quickly leveraged for traffic accounting purposes. This data could then be exported to collector software which would then extract and store the flow information, allowing for the collection of, and reporting on, network traffic data. Flow export technology is now well understood and has become widely used for security analysis, capacity planning, accounting and traffic profiling (Hofstede *et al.*, 2014).

Over the years, a number of newer technologies have emerged as alternatives to NetFlow. These include better known options such as sFlow¹ and IPFIX (Quittek *et al.*, 2004). sFlow differs from NetFlow in that it focuses on packet capture rather than network conversation tracking. The data captured can include a portion of the data (specified in bytes) from each packet that is sampled. Because the focus is on packets rather than conversations, there are often gaps in the sFlow output. IPFIX is an extension of NetFlow version 9 providing extensions that allow for template driven data export (Claise *et al.*, 2008). This enables implementers to add any number of new data attributes to the exported data. Cisco has leveraged this in its Flexible NetFlow technology² to allow for the inclusion of additional attributes such as packet data, deep packet inspection data, BGP policy counters, QOS data, etc.

Considering Cisco is NetFlow's biggest backer, and taking into account their continued dominance of the networking market (Trefis, 2017), it is safe to assume that NetFlow is

¹<http://www.sflow.org/>

²<https://www.cisco.com/c/en/us/products/ios-nx-os-software/flexible-netflow/index.html>

Table 2.1: NetFlow attributes

Core Attributes
IP source address
IP destination address
Source port
Destination port
Layer 3 protocol type
Basic Counters
Packets sent (source to destination)
Bytes sent (source to destination)
Additional Attributes
TCP flags
Interface identifiers
ASN numbers (usually only available in routers)
Source and destination netmask
Nexthop IP address
Type Of Service flag

still the most ubiquitous accounting technology around and can be found in all manner of connected devices. There are a number of different versions in use. The most common of which is version 5 according to Cisco Systems (2015a). NetFlow records, regardless of version, contain a number of common attributes (Cisco Systems, 2012). Table 2.1 lists the core attributes, the basic counters and some common additional attributes that may be collected.

Generally speaking there are three components in a NetFlow deployment as shown in figure 2.2:

Flow exporter This component resides on the device through which the traffic of interest flows. The export collects network traffic and aggregates the packets into flows. Network traffic of interest is captured, summarised into flow records and periodically exported to a flow collector device. On high-speed network devices a sample of the network traffic may be recorded and exported due to processing constraints.

Flow collector The flow collector is an application responsible for ingesting flow data from one or more flow exporters. The collector's primary function is the pre-processing and storage of flows. Some collectors may do minimal pre-processing, while others may implement more sophisticated routines (e.g. data enrichment, conversation correlation, etc.). As with pre-processing, the storage of flow data

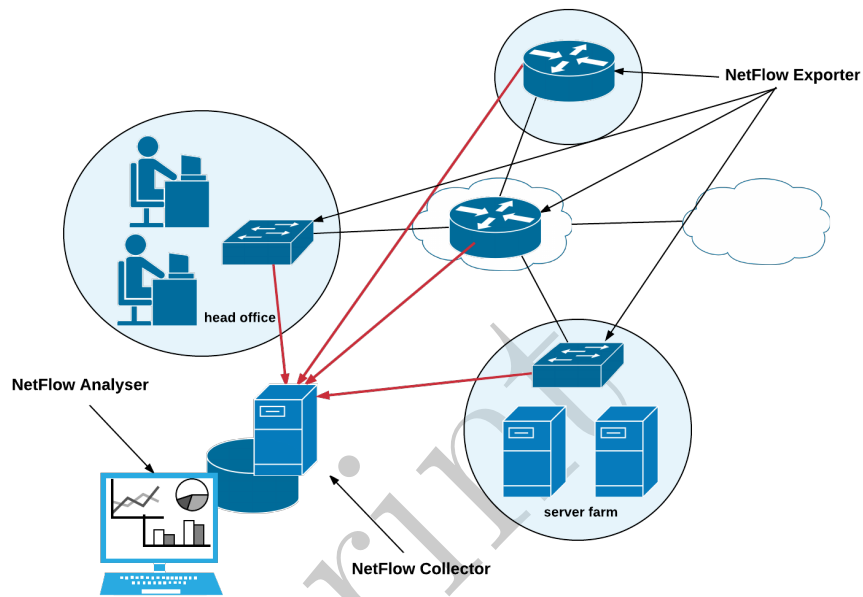


Figure 2.2: NetFlow deployment components

varies across collectors from simple binary files or databases all the way to complex structures designed for optimal querying purposes. Processing is traditionally done using batch processing approaches, but there is now a trend towards adopting Big Data approaches such as event stream processing or map-reduce technology in order to minimise the delay between capture and interpretation.

Analysis applications Collected flow data is analysed for a range of purposes. The majority of the time this is for network traffic accounting, but other applications are network performance measurement and intrusion detection.

For the purposes of this research, only a minimal subset of attributes is used during our analysis - the core ones identified in figure 2.3. This data is, however, to provide some key information about the network traffic exchange as illustrated in figure 2.3. We can tell who acted, when they acted and what they did from these attributes. Because this is the only type of data we will be processing, it makes the design of our data structures for ingestion and persistence straightforward.

2.1.2 Big Data and NetFlow Analysis

A crucial requirement of the design for the proposed real-time scoring framework is an understanding of the nature of the data that is going to be processing and its character-

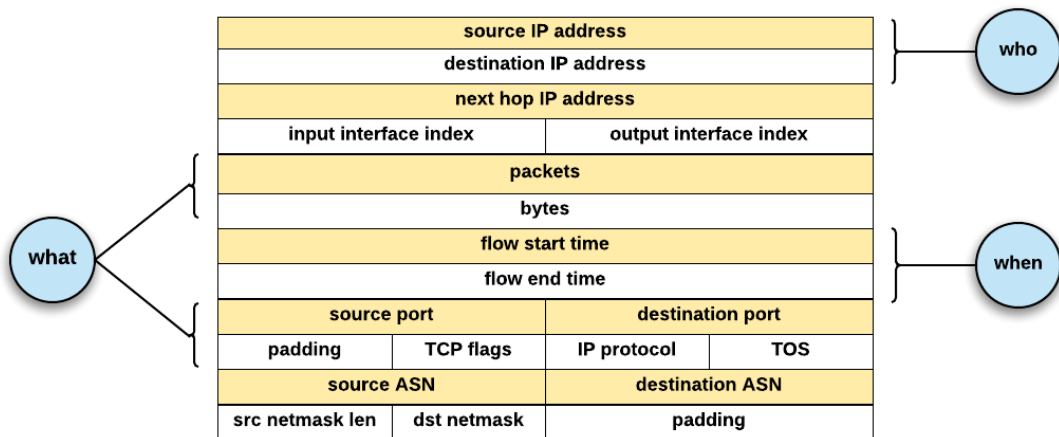


Figure 2.3: Core NetFlow attributes

istics. Along with this, the context within which we want to analyse the data plays a major role in guiding our architecture.

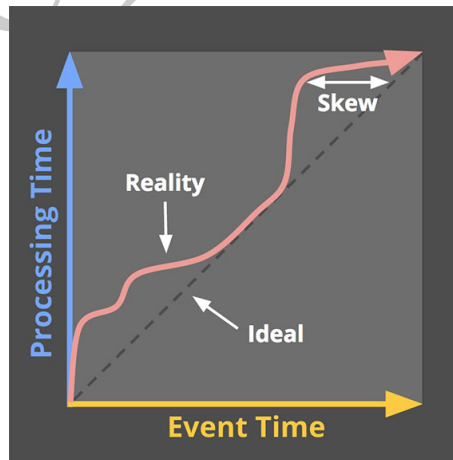


Figure 2.4: Event time domain mapping

Big Data is concerned with the processing and analysis of large data sets. What differentiates Big Data from traditional data analytics are three characteristics (Buhl *et al.*, 2013; IBM, 2013):

- **Volume:** More data than ever before is being created. It was estimated that in 2012 2.5 exabytes of data was created every day (McAfee and Brynjolfsson, 2012). As the volumes of data grow, tools and technologies are needed to scale efficiently in terms of costs and resources.

- **Velocity:** Along with an increase in volume comes an increase in the speed with which data is created. For some applications, the speed at which these volumes can be analysed is critical in order to leverage near real-time insights into the data.
- **Variety:** As the quantity has grown, so too have the sources and nature of data. One of the Big Data challenges is combining data from multiple data sources in multiple formats timeously in order to gain insight.

NetFlow by its nature matches the first two of these three characteristics. On large scale networks or when there are a large number of collection points, the volumes of data collected are significant. Additionally, in large networks, the data is generated at high volume, and in order to gain maximum benefit, needs to be processed at high speed.

There are three concepts from Big Data processing that are relevant to NetFlow data. The first concept relates to the domain of time and is the idea of event time vs. ingestion time vs. processing time (Apache Software Foundation, 2017). The first is the time at which an event occurred, the second is the time at which it was ingested into the system for processing and the last is the time at which the result of the event is observed in the system. In this context, observed refers to not just recordal, but also identification and classification of the event in such a manner that it can be reacted upon. The interval between event time and processing time is important as one of our goals is to maximise the value of our perishable insights by minimising this delay. This delay or skew is essentially the latency introduced by the processing pipeline and can be attributed to things such as processing resource limitations, surges in record throughput and algorithmic inefficiencies. The interactions between and the nature of these lead to a variance in the skew between event time and processing time over time as described in Akidau (2015) and illustrated in figure 2.4. Keeping the skew to a minimum is crucial in order to achieve as close to real-time latency as possible when processing the flow data.

The second concept of Big Data to consider is the idea of bounded vs unbounded data sets (Narsude, 2015a). A bounded data set is a finite set of data that is usually processed as a batch. As noted in section 2.2.1 most data is still processed in this manner following the traditional Extract-Transform-Load (ETL) cycle. Unbounded data on the other hand is an infinite, ever-growing stream of data. It is the second type into which NetFlow data falls squarely. Network traffic is an ever-flowing constant stream of data with NetFlow being an ever-flowing summary of this data stream. In order to maximise the freshness

of our insights when processing flow data, we need to apply unbounded data processing techniques to the data stream. This entails moving away from batch processing strategies to unbounded data processing ones where each data record is processed individually as close to the time that it was generated (near real time).

Taking the above into account leads us to the final concept that is applicable to Net-Flow data - the idea of data in motion vs. data at rest (IBM, 2013). For this context (information security), we need to treat the flow data as data in motion. The processing of data in motion requires different technology and processing strategies - data is analysed on the fly as it moves through a platform.

In summary, the nature of our data requires that our framework can analyse streams of flow data ingested at high volume with low processing latency. For the purposes of this work, processing implies the application of scoring and tagging as per Sweeney and Irwin (2017) and is described in more detail in chapter 3. Additional factors to consider are accuracy of the analysis and the perishable nature of insights in our context. This requires not only low processing latency (on the order of a few seconds), but also minimising the delay between event time and processing completion time.

A final consideration is data retention. Because we are dealing with an unbounded stream of data, due consideration needs to be given to the challenge of data retention. It would be impractical to attempt to store all the data received and so an archiving strategy would need to be considered. However, in our use case we are focusing on incident detection which implies that we are only interested in the latest data and if we miss something, it will be most likely too late to act on it. As such, the requirements for significant data retention could be for reporting (summarised data) or forensic (detailed data) reasons. We view retention policies and strategies as out of scope for this project.

2.1.3 Stream Processing

The technology that lends itself most to our problem domain is event stream processing (Čermák *et al.*, 2016). This section covers some of the concepts and terminology associated with it. The use of the word streaming can either refer to the data in question (streaming data) or the technology used to process such data. For the purposes of this discussion we

will refer to the data as an unbounded data set and the technology as stream processing. The core of the event stream processing paradigm is a sequence of components configured together in order to process incoming data streams. A key characteristic of these data streams are that they are an unbounded series of events ingested and transferred as messages through the engine for processing purposes. As illustrated in figure 2.5, there are two models for processing of the messages (Narsude, 2015b):

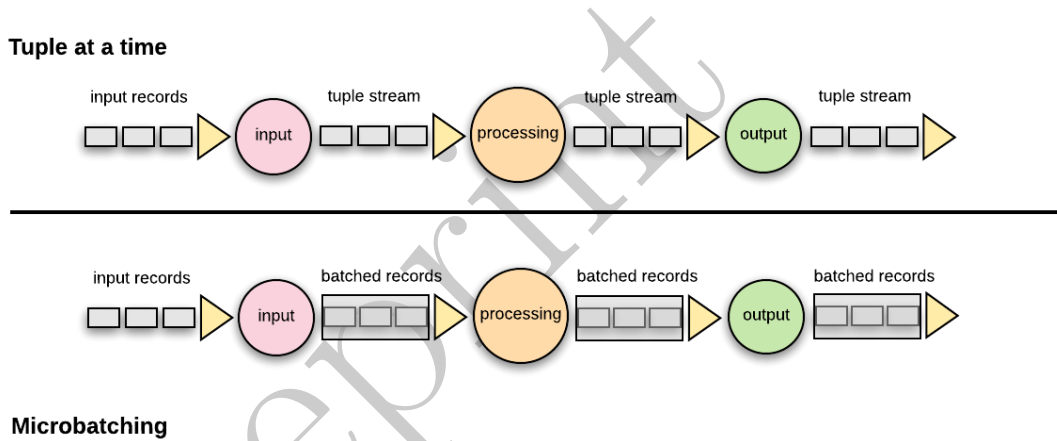


Figure 2.5: Message processing options

- **Tuple at a time:** In this model, messages are processed as they are consumed one at a time. This approach is the closest to the definition of event stream processing, but it can incur more overhead as the state of each individual message must be managed.
- **Microbatching:** This approach attempts to minimise state management by processing events in small batches (usually a few microseconds worth of events). The engine is only required to manage state for the batch of events. This approach does, however, add latency to the processing time.

The components, or building blocks, of an event stream processing are chained together in a directed acyclic graph (DAG) of components, each of which may perform discrete actions on the data. As illustrated conceptually in figure 2.6, the data flows from an input source, moves through the nodes in the DAG and is finally emitted after processing. Scaling, reliability and elasticity is achieved through spreading these units of work out over multiple CPUs and nodes. These fundamental building blocks perform the following types of functions on the streaming data (Cisco Systems, 2012; Akidau, 2015):

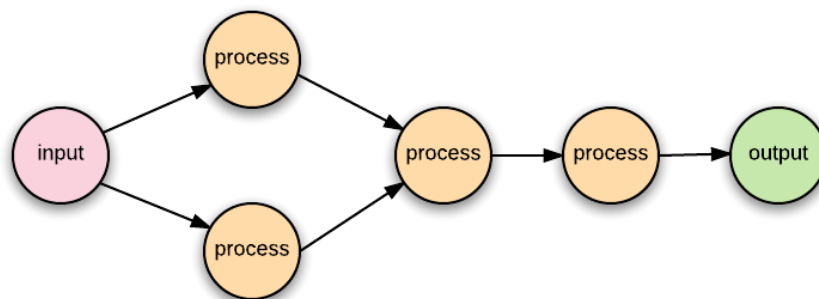


Figure 2.6: Stream processing DAG

Transformation Streaming engines ingest data from many different sources and in many different formats, components within the application may require only subsets of the event information or different data formats and finally, data outputs will have varying structural or schema requirements depending on what downstream systems will be consuming the data. These are all data translation requirements that are implemented in transformation components.

Enrichment Enrichment components draw on information in external data sources in order to add additional data attributes or context to the streaming messages. The extra information can be used by other components to assist in processing or it may ultimately be used in the final presentation or analysis of the results. Geolocation data is a good example of enrichment.

Correlation Data from multiple streams can be merged and the events correlated in order to aggregate events. An example of this would be matching proxy log information with network traffic logs. A complementary component would be one that split data streams into two or more new streams for different processing purposes.

Filtering Event streaming applications handle large volumes of data, large parts of which may be noise or irrelevant to the application in question. Filtering components allow for the manipulation of data streams in order to extract the information that is pertinent to the question at hand and discard irrelevant data.

Windowing Stream processing is about processing flows in real time, but often applications need a snapshot of the data over an arbitrary period of time. This is referred to as temporal windowing (a form of micro batching) and is a requirement in many stream processing applications (Akidau, 2015). Examples of this are summing the total number of visitors per minute or calculating the average number of clicks in

the last five minutes. The most common types of windows can be one of *sliding* (all data in the last five minutes), *tumbling* (snapshots of data every five minutes) and *fixed* (e.g. from 10:00 to 10:05). For our purposes we will require windowing support when analysing flows for traffic patterns that require the examination of a number of conversations (e.g. port scanning).

Logic Components The aim of stream analysis is to identify insights and possibly react to them with real-time context. Logic components can be used to evaluate the processed events and publish the derived information to external applications for notification, visualisation or analysis purposes.

In the Themis framework the components that perform the scoring of the flows are a special case of enrichment processing, as the scores and tags can be considered an enrichment of the flow records. These components should be implemented as a collection of stand-alone independent components, each of which performs one and only one scoring test. This preserves the concept of single responsibility and allows us flexibility in the design of our DAG processing topology. The event stream that is processed will consist of an unbounded stream of flow records.

Ensuring completeness when processing bounded data is well defined - if there is a failure, then the processing can be restarted on the same set of data. The bounds of the data are well defined and with processing checkpoints recovery from a failure can be speeded up (if a batch fails to process, it fails in its entirety and it can be resubmitted). With stream processing there are challenges introduced by the message oriented nature of the processing model: nodes can fail, queues may fill up, components can time out and messages can be lost in transit through the platform. For the completeness and accuracy of our processing we need to understand the possible message delivery guarantees of the streaming technology chosen. There are three possibilities (Tzoumas, 2015; Narkhede, 2017):

- At most once: Lost messages are never resent.
- At least once: Messages are never lost, but they can be redelivered.
- Exactly once: Messages are only ever delivered once.

The last option is the most difficult to implement and most solutions will provide an at least once guarantee (Narkhede, 2017). The design of Themis needs to take the delivery guarantees of the selected streaming engine into account in order to address accuracy and completeness concerns.

2.1.4 Flow Scoring

Flow scoring is an approach wherein a series of independent tests are applied to flows. If the test criteria is met, then this outcome is used to assign a score to the flow (Sweeney and Irwin, 2017). Depending on the nature of the test, this score can either be a good or a bad score. Goodness is a measure of how normal or expected the traffic in question is (for example YouTube traffic), while badness is an indication of suspicion (e.g. port scanning, brute force SSH attempts, etc.). The resulting cumulative good and bad scores are purposefully kept as two separate scores in order to allow the differing nature of the flows to be separately recorded and evaluated. An important feature of this approach is that the flows are scored rather than the hosts themselves. An individual host's reputation may contribute to a flow score (for example, if it appears on a blacklist), however, the scores are still applied to the flows as it is the traffic between hosts that has the good or bad traits. As part of the scoring process a flow will also be tagged if the test criteria are met. This tagging serves to add a layer of metadata to the flows for analysis purposes. The benefits of the tagging are threefold: the reasons for a high or low score can be identified from the list of tags applied to a flow, tags can be used for filtering flows (inclusive or exclusive filters), and finally, they provide a means of summarising the flows by their nature.

2.2 Literature Survey

In this section, selected background from the literature is presented. Three different areas are covered: the use of NetFlow in information security, the application of Big Data to information security (in particular NetFlow), and finally, scoring approaches to incident detection.

2.2.1 Using NetFlow for Security Incident Detection

Netflow has been found to be especially useful for the detection of DoS attacks, network scans, worms and botnets (Hofstede *et al.*, 2014). In addition, because organisations typically store flow data for some period of time, the flows can assist in forensic investigations. The commonality of these attacks is they all affect network metrics such as flows, packet counts, byte counts, etc. But analysing these attributes alone is not enough to give insight into the attacks an organisation may experience. In this section we review work that has been done in order to gain insight into what approaches may be applicable to our analysis.

A common theme in prior research has been the use of blacklists and whitelists to preprocess the data (see Vaarandi and Pihelgas (2014), Hofstede *et al.* (2014) and Amini *et al.* (2014)). Blacklists are lists of IP hosts suspected to be involved in malicious behaviours such as acting as botnet Command and Control nodes, known compromised nodes, known port scanners, etc. Often the blacklist may include reliability or reputation scores to give an indication of confidence in the host's perceived threat. These lists are compiled and maintained by numerous security institutions such as Emerging Threats³ and AlienVault⁴ and are made available as regularly updated feeds. Whitelists are lists of IP hosts that are known to be reliable or common destinations that are not likely to be a risk. Examples of these are Google, Facebook and Amazon. The suggested approach is that flows with blacklisted hosts are marked as suspicious, while flows with whitelisted hosts can be excluded from any further investigation.

The use of traffic profiling for anomaly detection is discussed in Choudhary (2013) and Vaarandi and Pihelgas (2014). This approach requires profiling either host or network traffic over a period of time and then using various methods for the detection of anomalies. This, however, requires data that has been collected over a significant period of time in order to establish a baseline against which to look for deviations. Similar approaches look for network patterns on a much smaller scale. For example, scan detection can be done by counting distinct connection attempts made by a source within a particular time interval (Chandrashekar *et al.*, 2015). The performance and accuracy of these approaches depend on the time interval chosen. Probabilistic models have been used to try and improve this approach. Simple methods for detecting the spread of worms have also been used such as

³<https://rules.emergingthreats.net/>

⁴<http://reputation.alienvault.com/reputation.data>

looking for unusual top-N connections or top-N bytes within a short time interval (Khule *et al.*, 2014).

Flow data contains the TCP flags from the network conversation which allows for simple analysis to detect DoS attacks or scans. Choudhary (2013) describes the detection of DoS attacks by looking for unusual traffic targeting single hosts that has only the SYN or RST flag set. Large numbers of flows with only the SYN flag set are typically the result of port scanning and can be recorded and reported on (Amini *et al.*, 2014). Finally, flows with illegal flag combinations (e.g. TCP FIN without an ACK) can also be detected and marked for further investigation (Vaarandi and Pihelgas, 2014).

A simple method of reducing the number of flows that require investigation is the filtering out of traffic to known hosts and ports (Vaarandi and Pihelgas, 2014). An example of this is for traffic to and from port 25 on a known mail server to be excluded from the dataset. Conversely traffic to and from unknown ports or servers can be flagged for further investigation.

ICMP flow data can be used to detect scans or malicious connection attempts such as those caused by worms (Khule *et al.*, 2014). High numbers of ICMP destination unreachable or ICMP port unreachable messages to a host or hosts may indicate scanning or malicious connection attempts on a network.

In Hofstede and Sperotto (2014) the authors propose an attack detection methodology whereby they compare flow characteristics to expected behaviour. In particular, they consider brute force SSH attacks. By modelling the network conversation traits of a failed SSH login attempt (i.e. session setup, number of login attempts, etc) they are able to reliably detect brute force attacks by looking for large number of SSH flows which only have between 11 and 51 packets. In Krmicek (2011) the authors propose a similar approach for the detection of botnets. By taking into account the characteristics of a botnet and the related Command and Control (C&C) activity, they propose monitoring for patterns that could indicate a potential infection. For example, observing a large number of DNS requests from many hosts at the same time followed by other synchronised network traffic from the same hosts could be a strong indicator of the existence of a botnet on a network.

A number of approaches using scoring of hosts based on flow attributes for automated threat detection have been proposed. As early as 2004 the MINDS system was proposed with the aim of solving the dependency in pre-knowledge for intrusion detection (Ertöz *et al.*, 2004). At the time threat detection relied heavily on threat signatures and manual intervention. The MINDS system aimed to solve this problem through the automatic assigning of an anomaly score to network connections. The higher this score the more likely the connection was suspect. High ranking connections could then be assigned to a network operator for further investigation. In Marchetti *et al.* (2016), the authors propose using a similar approach to scoring hosts on a large internal network in order to detect APTs. They apply various statistical algorithms that use flow attributes (specifically, the number of bytes sent, number of flows and number of destinations) to score a host over time with a specific focus on detecting data exfiltration. Commercial vendors such as Cisco's IronPort have also implemented host scoring for threat detection and applications that can work with external reputation feeds to aid in scoring and alerting (Schiffman, 2012).

2.2.2 Big Data and Network Security

In section 2.1, the nature of NetFlow data was discussed and contrasted its characteristics with those of Big Data. In section 2.2.1, we looked at the applicability of using NetFlow in a security event detection context. In this section we bring the two concepts together and review some work done in bringing Big Data technologies and concepts into the world of security and in particular the application to NetFlow analysis.

Some of the first applications of Big Data to the processing of NetFlow data were done primarily for traffic analysis. For example, in Lee and Lee (2012) the authors describe a system using Hadoop to process IP flow statistics. By applying MapReduce algorithms on a 200-node cluster, they were able to achieve a throughput of 14 Gbps. While primarily focusing on network statistics analysis they also implemented simple DDoS traffic detection using this system. All processing was done batch style in an offline mode but they state that real-time processing would be feasible using this approach. In Bar *et al.* (2015) the authors describe a custom stream processing solution for processing network data by treating it as data streams for real-time processing. Their solution is not true stream processing but rather a form of micro-batching. Incoming log or traffic data is processed at high speed in small batches. They have, however, implemented a number

of the processing primitives found in stream processing frameworks such as aggregation, filtering, etc. What is particularly interesting about this work is they claim their network outperforms Spark. While this may be the case, they have had to implement a lot of the functionality needed from scratch. In addition, the application relies on PostgreSQL⁵ for most of the processing which brings in the common problems of RDBMS scaling. With respect to scaling, Gupta *et al.* (2016) present another custom solution for processing high volumes of network traffic by filtering what is collected at source and then sending this traffic to a stream processing system based on Apache Spark⁶. The motivation for this approach is their belief that while stream processing systems are able to scale out as the load grows, the amount of data generated by large backbone networks and Internet exchange points far exceeds the capabilities of these systems. The filtering leverages a Software Defined Networking (SDN) platform by implementing their custom filters in the data plane on the switches, thus vastly reducing the amount of data that has to be exported for analysis. The programmable nature of the SDN switches allows the filtering rules to be updated as and when required.

Pure event stream processing of flow data has been documented in the literature a number of times. As early as 2014 the authors of Du *et al.* (2014) presented work wherein they built a system leveraging Apache Storm⁷ to process flow records and look for anomalies in real time. Their solution uses statistical methodologies and was able to process just over 30,000 flows per second on a five node cluster (two CPUs per node). Similarly, Lee *et al.* (2015) describe a Storm and Hadoop system that does statistical analysis of connection attempts and breaks out unknown services vs. unknown connections vs. abnormal connections. Their system was tested on a relatively small control data set of 1.5 million connections that contained a known number of malicious flows. Machine learning is coupled with stream processing and other Big Data technologies in Zhao *et al.* (2015) for real-time security event anomaly detection. The work implements three bolts in Apache: one for data pre-processing, one for anomaly detection and finally, a machine learning bolt that looks for patterns in the anomalies. The system has been able to process up to 8,000 flows per second with this solution on a five node cluster (each node comprising a four core CPU). The processing numbers are not as good as some of the other work described in this section. This is most likely due to the introduction of machine learning to the mix. The overheads of this would need to be weighed up against the benefits.

⁵<https://www.postgresql.org/>

⁶<https://spark.apache.org/>

⁷storm.apache.org/

Finally, in Jirsik *et al.* (2017) a stream processing system is described with many similarities to the design presented in this work. The authors identify a number of advantages of stream processing flow data for threat detection: streaming enables analysis of flows immediately after observation, the amount of data that needs to be stored is reduced and the load can be distributed amongst many loads in a cluster. In addition, the low latency of the processing time gives users immediate data analysis capabilities making near real-time threat detection possible. The system they built was based on Kafka⁸, Apache Spark⁹ and Kibana¹⁰ and on a 32 virtual CPU cluster it was able to process to two million flows/second. Their analysis consisted of a number of MapReduce jobs for calculating host and flow statistics. The conclusion of their work was that this architecture represents a natural complement to current batch-based approaches for cyber security allowing for high throughput, low latency solutions that can easily scale. A survey of a number of different approaches using Big Data technology is presented in Wang and Jones (2017) along with a description of their own implementation of an unsupervised anomaly detection approach using Spark and machine learning. Their analysis focused on data sets containing known botnet traffic and they were able to detect the botnet traffic with 96% accuracy. In their survey the authors summarise a wide variety of solutions that combine different Big Data technologies and analysis methodologies most of which show high success rates.

Čermák *et al.* (2016) proposed and tested a novel benchmark for testing performance of event processing systems with respect to NetFlow data processing. Their goal was to not only test throughput and latency, but also to test with meaningful processing. They therefore implemented a number of processing operations that would typically be used for NetFlow analysis. This included filtering, aggregation, TopN counts and SYN DoS detection. Their benchmarking covered three common streaming platforms: Storm, Spark and Samza¹¹. While each of these are stream processing systems, they vary in things like programming language, parallelism, message processing, etc. In all three cases, they were able to demonstrate processing 500,000 flow/s using 16 or 32 processor cores. Samza delivered the highest throughput, but at the cost of some inflexibility. Storm was the worst performing system, but was still able to achieve peaks of up to 1,000,000 flows/s in the test environment. While informative in terms of throughput testing and scaling, the work did not look at the total throughput of multiple operations, but focused

⁸<https://kafka.apache.org/>

⁹<https://spark.apache.org/>

¹⁰<http://www.elastic.co/products/kibana>

¹¹<http://samza.apache.org/>

instead on benchmarking individual ones. A key statistic that was missing was the latency measurements. High throughput does not necessarily equal low latency. Despite the poor performance of Storm, its selection for use in this research was deemed suitable as the benefits of a simpler programming and processing model outweighed any potential performance gains.

In this section, a brief look was taken at the application of Big Data technologies to NetFlow processing. The first applications were for network statistics with a logical consequence of this being to look for network anomalies and then security related issues. The flow analysis done was mostly statistical with machine learning becoming more popular in recent years. A number of common themes was present across the various papers:

- The amount of network data that is available for inspection is becoming a bigger and bigger challenge.
- Big Data technologies appear to be well suited to the processing of this data.
- Big Data frameworks allow for the application of different analysis approaches at scale.

There appears to be a consensus on the applicability of Big Data technologies for flow data processing and, in particular, for security event detection. This lends weight to the proposed approach, giving confidence that there are solid foundations on which to proceed.

2.2.3 Scoring for Network Security

One of the core concepts of the proposed work is the idea of scoring the flows in order to raise the profile of suspect traffic while allowing for the filtering out of known (or good) traffic. The idea of using scoring in security has been around for a while, most notably in anti-spam applications such as SpamAssassin¹². Originally SpamAssassin was implemented as a Perl script that implemented a number of simple rules, each of which had a score associated with it. When a certain score threshold was exceeded, the email in question was deemed to be spam. Over the years the mode of operation has evolved into

¹²<http://spamassassin.apache.org/>

using a combination of many different approaches, including neural networks, blacklists and DNS scoring, but the principle of the scoring has remained. Another feature of the scoring operation is the idea that there are both good scores and bad scores which together make up a final measure of how likely it is that the email is a spam message. There is some work that has been carried out which applies scoring to NetFlow analysis. Two of the more relevant ones are reviewed below.

While not true scoring, a system using a variant of the Google PageRank algorithm to detect botnet traffic was proposed by François *et al.* (2011). The framework takes flow data and builds a host dependency graph to track communication patterns. The graph is then analysed using Google PageRank to identify nodes that are strongly linked. Additional information from honeypots is used to supplement the data. The authors have applied the technique to traffic data from a large ISP and have determined that this approach is viable.

Marchal *et al.* (2014) describe a system that takes flow data and, using Big Data frameworks, applies a scoring strategy based on four different primary sources: DNS records, HTTP requests, IP flows records and honeypot data. In addition, third party data such as blacklists and techniques from prior work are also applied. The result is three scores relating to DNS, Web requests and IP flow data. These are evaluated individually as computed for alerting, as well as in combination at the end of the scoring process. The scoring process was implemented on a number of different frameworks and tested throughout, with Apache Spark coming out tops in terms of performance. All work was done as offline analysis with no testing of real-time processing. Future work on this aims to move to online analysis using Spark Streaming or Storm.

Scoring as a technique has been tried and tested in areas such as spam detection. A number of researchers have taken this approach to flow analysis with promising results. The work of Marchal *et al.* (2014) appears to be the most similar to this project, however, where they applied a limited number of tests and were doing offline scoring the goal of this work is to carry out real-time scoring with a more generalised approach to scoring. One of the areas of concern with many studies discussed in this chapter is the use of well-known, static data sets containing previously identified and labelled incidents. The problem is that building systems to detect attacks using labelled data runs the risk that the system will not be able to adapt to unknown or new attack patterns (Zuech *et al.*,

2015). The proposed scoring approach laid out in this work deals with this in a number of ways. It is able to leverage tried and tested methods, it can look for anomalies in expected traffic and methods can be implemented to look for suspicious behaviour. In all cases the matching of flow data to a test is not an absolute indication of a threat, but rather contributes towards evidence thereof.

2.3 Similar Implementations

There are small number of open source and commercial products that mix Big Data with NetFlow analysis. These are discussed briefly in this section for completeness.

PacketPig (Arbor Networks) is an open source project that takes packet captures and device logs to be stored, analysed and visualised. It is built using Apache Pig¹³, a data analysis platform that runs on top of Hadoop. The platform uses a number of customer data loaders to classify traffic including traffic fingerprinting (using p0f¹⁴, HTTP deep packet inspection, Snort inspection, etc). Other than at load time, little analysis is done and the functionality being more geared around visualisation of potential security issues.

OpenSOC (OpenSoc, 2014) is one of the more well-known open source Big Data flow processing solutions. It started out as a project at Cisco in 2014 (Cisco Press, 2014) and a number of articles have been written documenting its achievements, most notable of which is reaching a processing throughput of 1.2 million flows per second (Sirota; Sirota and Dolas, 2014). OpenSOC is built on top of a number of Big Data frameworks including Storm, Hadoop, Kafka and HBase. It also makes use of MySQL for persistent storage and Elasticsearch/Kibana for indexing and visualisation. The framework has extensible components for adding telemetry data to flows, for implementing rules or anomaly based alerts and integrations with existing analytics tools.

In 2015, Cisco stopped supporting OpenSOC after open sourcing it (Salazar, 2014). From there the project evolved into the Apache Metronome project (Apache Organisation, 2016). Metronome is described as a “cyber security application framework that provides organisations the ability to ingest, process and store diverse security data feeds at scale

¹³<https://pig.apache.org/>

¹⁴<http://lcamtuf.coredump.cx/p0f3/>

in order to detect cyber anomalies and enable organisations to rapidly respond to them”. The framework ingests data (called telemetry events) from multiple sources to process, enrich and label it. This output can be used for alerting, data modelling, analysis and visualisation. Enrichment includes the addition of geolocation tags, autonomous system information and WHOIS data. In the labelling phase, the telemetry events are labelled through either scriptable actions or through data sourced from third parties such as threat Intel feeds. Finally, alerting and persistence takes place. This data is then made available for further analysis or modelling. As with OpenSOC, Metronome is built around Storm while using similar technologies such as Hadoop, ElasticSearch and Kafka.

The examples in this section show an increasing sophistication in the application of open source Big Data tooling to the security challenge. The framework proposed in this document bears a striking resemblance to that of Metronome but has a number of key differences. The biggest difference is the scoring concept - Metronome applies a tag or label to events and then reacts or reports on that. The scoring approach applies multiple tags along with scores (effectively weights), thereby allowing for a more layered approach to threat detection. In addition, the idea of good vs. bad scores enables filtering out of known traffic or the contrasting of the known vs. the unknown.

2.4 Summary

In this chapter a technical background is presented in section 2.1 on NetFlow, Big Data concepts and stream processing. This serves to provide the reader with the context for the research being presented and paves the way to an understanding of how NetFlow processing lends itself to being a Big Data and, in particular, a stream processing problem.

In section 2.2, related work in the field of NetFlow security analysis, Big Data in security and scoring for security purposes is presented. Finally, some open source frameworks following the same approach are discussed in section 2.3. The reviewed literature gives further weight to the work proposed in this document with a number of similar frameworks or architectures having being well documented and tested. Ideas and concepts discussed in this chapter are applied in the following chapter in order to specify a design and architecture for Themis.

Chapter 3

Themis Design, Architecture and Requirements

In order to meet the goals of the project, the first step is the design and architecture of the Themis framework. Informing our decisions on choices of tools, components and architecture are our two project goals as set out in chapter 1. To borrow a concept from the world of hardware design, we can approach our design in terms of micro-architecture and macro-architecture. The first, micro-architecture, relates to how we implement a componentised means of flow scoring and tagging. The second, macro-architecture, speaks to how we design the overall framework within which the NetFlow data can be ingested, enriched, scored, filtered and processed, and the results exported for interrogation or visualisation. An overarching guiding principle to both of these is that we need to aim for a solution that can scale to high volumes, while maintaining low throughput latencies. The implementation of the design is discussed in chapter 4.

This chapter documents the architecture and requirements of the framework as informed by the background presented in Chapter 2. Section 3.1 outlines the architecture of the Themis framework, section 3.2 describes the detailed requirements and in section 3.3 the section is summarised.

3.1 Architecture

As discussed earlier, the framework can be considered to have a micro- and macro-architecture. In this section these proposed architectures are presented. In figure 3.1

the conceptual architecture is presented. This illustrates the proposed design at a very high level, but clearly delineates the major functionality required.

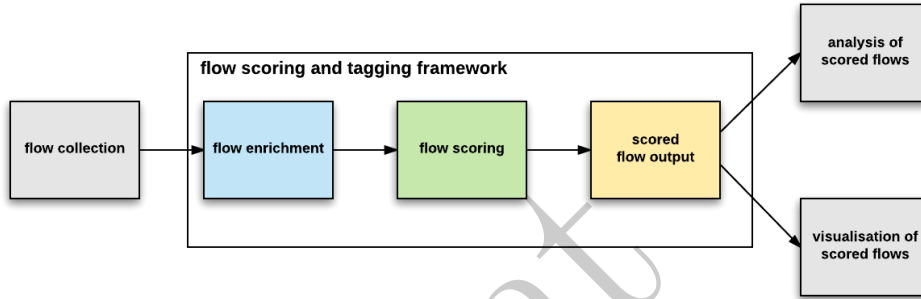


Figure 3.1: Themis conceptual architecture

3.1.1 Macro-Architecture

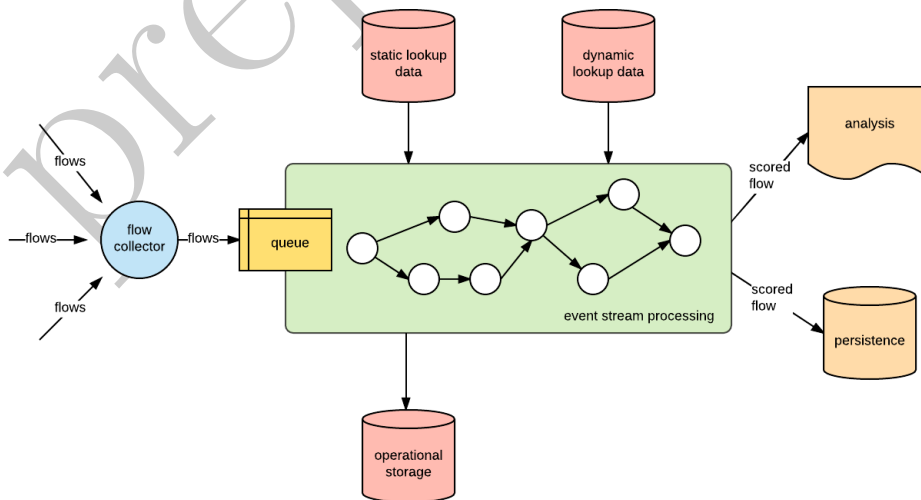


Figure 3.2: Themis high-level architecture

Our use case for processing the flow data fits the event stream processing model of data processing. Our architecture is therefore based on the Kappa Architecture, a real-time data processing architecture ideally suited to event stream processing (Pathirage, 2017). Our realisation of this architecture is illustrated in figure 3.2. The starting point for our data processing is to be able to ingest flows from multiple sources. Ingestion of data in a Kappa Architecture is done through the use of a high-speed publish-subscribe messaging or queuing system. This decouples the flow collectors and the processing engine, and allows for the ingestion of data from multiple sources.

The core of the Themis framework is the Stream Processing Engine (SPE). This component will do the actual processing of the flow data through the chaining together of discrete components each of which performs a single task on flows as they are streamed through the engine.

In order to support the processing, the stream processing engine will require access to external dynamic and static data sources. In addition, the engine will require an operational data store for temporary persistence of flow information during processing. Finally, our design must cater for multiple outputs of our scored data. The data must be made available to for both real time analysis and visualisation as well as persisted in a database for recordal purposes or further analysis. In addition, the design must allow for the persistence of the score flows for further analysis or audit purposes.

3.1.2 Micro-Architecture

The micro-architecture describes the sequence of processing of the individual flows as they move through the stream processing component. This represents the intended pipeline that flows will be streamed through inside the stream processing engine in order to be scored and tagged. Once processed the flows are outputted for visualisation or persistence. Non-transient data should not be stored in the core of the framework.

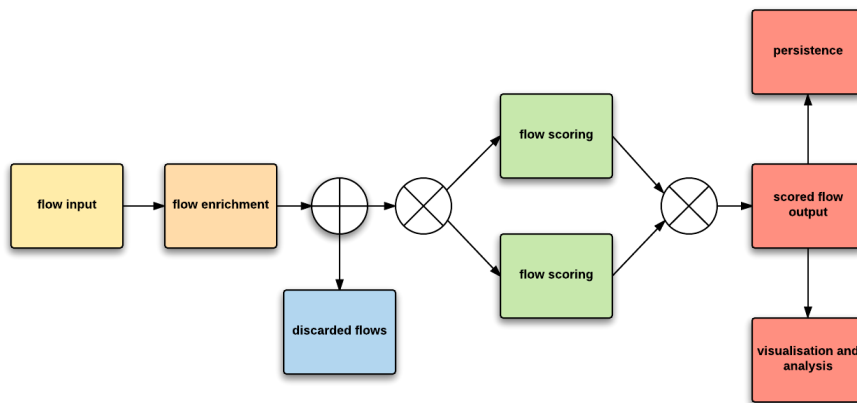


Figure 3.3: Stream processing architecture

In the diagram the boxes represent groups or classes of components (flow processing components) that will be processing the flows. Each component in the streaming engine

must be implemented as an independent entity able to be scaled up through multiple instantiations. The implementation of the stream processing engine must allow for flexibility in the construction of the scoring pipeline to cater for different scenarios. An additional requirement is that the scores and tags assigned in each component should be configurable.

3.2 Themis Requirements

In this section we outline the detailed requirements for the design and implementation of Themis. These requirements are developed from the design goals, the concepts and the architecture design. They must define a set of features that will inform the implementation process and the choice of technologies used.

3.2.1 Data Collection

For the purposes of this work the NetFlow collection will be assumed to have been done using either commercial (e.g. Solarwinds¹ or Scrutinizer²) or open source tools (e.g. flow-tools³ or cflowd⁴). The only requirement will be that there exists an API or utility that allows for the extraction of flow data in a format that can be transformed into a standard input format for processing (this format is described in chapter 4). This format must include, at a minimum, the seven basic NetFlow attributes described in section 3.1.1, along with a unique identifier for the flow record. In addition, a number of other attributes relating to the flow time are required. The full set of minimum requirements are listed in table 3.1.

3.2.2 Data Ingestion

In order to ingest the flow records created by the collector, a utility needs to be implemented that can extract the data from flow collection tools, transform the data into a standardised format for consumption by the system, and then submit it to the framework

¹<http://www.solarwinds.com/netflow-traffic-analyzer>

²<https://www.plixer.com/products/scrutinizer/>

³<https://github.com/adsr/flow-tools>

⁴<http://www.caida.org/tools/measurement/cflowd/>

Table 3.1: Minimum flow attributes

Attribute	Type
Unique Identifier	Integer
Flow Start Time	Unix epoch time
Flow End Time	Unix epoch time
Flow Duration	Seconds
Source IP Address	Dotted quad notation
Destination IP Address	Dotted quad notation
Source TCP Port	Integer
Destination TCP Port	Integer
TCP Flags	String
IP Protocol Number	Integer
Packets Sent	Integer
Bytes Sent	Integer

via a messaging queue. This allows for high speed, asynchronous submission of data from multiple sources in a decoupled manner. In order to ensure a minimum skew time for data processing, the queuing platform should be a high-throughput, low-latency platform.

3.2.3 Flow Enrichment

For enrichment of the flow, the following flow processing components are required (this is represented by the second block in figure 3.3):

- **IP Geolocation:** the source and destination IP addresses in each flow should be mapped to the source locations. This mapping should be done through locally cached databases rather than through remote API calls for optimal lookup times and reduced network overheads.
- **Autonomous System Numbers:** the source IP and destination IP addresses in each flow should be mapped to the respective Autonomous System Numbers to which they belong. This mapping should be done through locally cached databases rather than through remote API calls for optimal lookup times.
- **Bidirectional Traffic Correlation:** NetFlow data is typically collected as unidirectional flows with two records being captured and stored for each network conversation. Themis should, where possible, match up the flows that make up a conversation and combine the bidirectional traffic data (bytes and packets) from both flows into a single conversation.

The data field requirements for the information added by the enrichments must be considered when implementing the data structures used in the framework and in any persistence layer. The implementation of the enrichments is discussed in detail in chapter 4.

3.2.4 Flow Scoring and Tagging

The concept of scoring data for analytics or decision making is not a new one (see section 2.2.3). In finance, credit scores are generated using a multitude of different criteria (FICO, 2017), SpamAssassin uses a scoring system to detect and flag spam (Apache Software Foundation, 2009) and there are a number of services offering IP or mail MTA reputation scoring services (Sender Score, 2017)). For the purposes of this project we need to discuss what the scoring requirements are for the flow data. We should also demonstrate a variety of scoring methods in the design and implementation of the system. The following list of items broadly covers the range of options that need to be supported by the scoring components described in the following sub sections.

- **Data Enrichment:** While not strictly a scoring function, enrichment is required in order to assist in the scoring process, for visualisation, for filtering and for analysis of the processed data. Enrichment can include the addition of geographic information, ISP autonomous system numbers (ASNs), matching unidirectional flows, etc.
- **Static Lists:** Scoring using information from static configuration makes use of statically defined configuration such as interesting ASNs, dark IP address space, port numbers, blacklists and whitelists. Static lists may be made up of multiple attributes, for example IP address and port numbers.
- **Dynamic Data:** This entails sourcing constantly changing and updating information from security feeds for scoring purposes.
- **Signatures:** As part of the framework we wish to demonstrate that we can leverage prior work in order to identify threats. This is demonstrated through the use of components that can score potentially malicious flows using traffic signatures identified and documented by other researchers.
- **Heuristics:** Similar to signatures, prior work has identified heuristics that can be applied in order to identify malicious traffic. Additionally, we may wish to apply some common practices in our scoring (e.g. any traffic between unreserved ports is suspicious).

- **Context Specific Scoring:** The majority of scoring will be done on a per record basis, however, some scoring will require context. By context, we mean the scoring logic needs to review a set of records in order to determine if they match specific characteristics. The most common example of this is detecting port scanning. Scanning detection cannot solely be done in isolation (e.g. looking at a single record), but requires looking at a subset of data over a small time window.
- **Anonymising:** A final step in the flow processing is the anonymisation of the data for reporting and privacy purposes in this research project. For this, only IP addresses on the inside of the network where the NetFlow was collected need to be anonymised. The ASN in question will also need to be anonymised. In real world or production deployments there is no need for this component unless there is a requirement to share the data with third parties.

The list of different scoring scenarios presented by no means covers all the variations possible, but it does present a significant number of different options that can be used to demonstrate the effectiveness of Themis. Any scoring or tagging values linked to tests should be configurable, allowing the values to be changed easily in order to experiment with different scoring outcomes.

Following on from these requirements, the framework must be able to meet the following two goals for scoring and tagging:

- Scoring and tagging components must be independent units of work that, where possible, are not dependent on other components in order to function. This excludes special cases where components which may require enrichment information in order to effect score and tagging (e.g. scoring that is dependent on knowing the geolocation of the hosts in the flow).
- Themis must provide flexibility in terms of the sequence in which the scoring is executed. This should allow for parallel processing, different work streams, filtering of flows and merging of results. This flexibility is required for the processing to be adapted to differing workloads and and varying scoring component combinations.

The scoring and tagging of flows also requires that the data structures used in the implementation can store the values as required. As discussed previously, our requirements are that we store good and bad scores separately (see figure 3.4). We will also need to store the good and bad tags in separate lists along with the scores associated with each one. This information will assist in the experimentation and analysis.

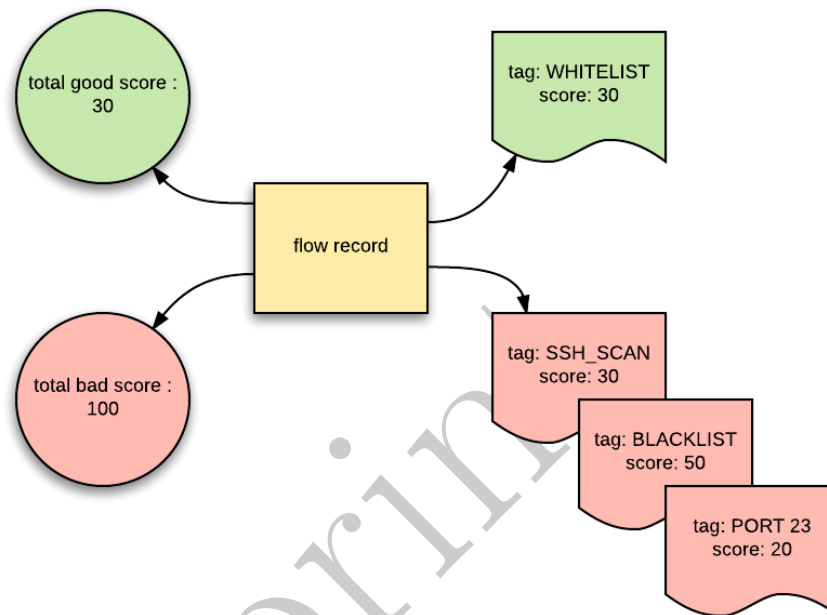


Figure 3.4: Flow scores and tags

The following scoring and tagging enrichment components are required to be implemented in order to demonstrate the feasibility of Themis (these are represented by the blocks labelled “flow scoring” in figure 3.3):

IP Address Based Scoring A component for scoring of a flow based on the presence or absence of the source or destination IP address in a list.

Port Based Scoring A component for scoring of a flow based on the presence or absence of the source or destination port in a list.

IP address and port combination scoring a component for scoring of a flow based on the presence or absence of the source or destination IP address and port combination in a list.

Threat Intelligence Scoring A component that can score flows using information from multiple external threat intelligence sources.

Signature or Heuristics Based Scoring At least two components should be implemented showing scoring using methods described from other related research.

Scan Detection Components are required for the detection of horizontal and vertical scans.

The implementation of components should be generic where possible. For example, the list based components should be configuration driven, allowing multiple lists to be used in different instances of the same component. The list of components in this section form an initial implementation of our approach, but can easily be added to or extended through further work.

3.2.5 Utility Components

A number of utility components are required in order to facilitate the flow scoring process. These components will assist in the design and implementation of the scoring DAG. The following are requirements at the very least (these are represented by the circles in figure 3.3):

- **Stream Splitting:** We may want to split up the stream of record as they pass through the processing engine. Reasons for splitting include: processing different protocols in different streams, different categories of scoring and to leverage parallelism (i.e. executing a variety of scoring processing at the same time).
- **Merging of Streams:** Along with splitting, we require merging components (correlation component). This requires consuming multiple streams of records with no fixed order and ensuring that individual flow records arriving on different streams at different times are merged back into a single flows record that can be sent out on a single stream.
- **Instrumentation:** For tracking performance and reliability, a number of instrumentation components are required. These will be used for measuring record flow latency and throughput.

3.2.6 Output Components

Scored flows need to be outputted from the streaming engine for analysis and visualisation purposes. These are variations on transformation components of which a number of options are required:

- **Database Persistence:** Scored flows must be persistent in a database for analysis and recordal purposes. This can either be a traditional RDBMS or a NoSQL document store.

- **Output Queue:** Scored flows must be sent to an output queue for consumption by visualisation and alerting systems. By sending these to a general queue, multiple options are available for action upon on the enhanced flow records.
- **Logging:** For testing and debugging purposes a general logging component is required that will output flows to a standard logging framework that can be stored and accessed at any time.

3.2.7 Configuration

For testing purposes some level of configurability is required in the scoring framework in order to evaluate if the project goals have been achieved. The extent of configurability required needs to cater for configuring the following behaviours at a minimum:

- **Score Points:** The values assigned to flows by the different scoring components should be configurable in order to enable changing the weighting of the tests or categories of tests. This will enable different scenarios to be run evaluating the effectiveness of different scoring approaches.
- **Score Labels:** The labels used for flows in the scoring components needs to be configurable. The labels will be used in reporting and visualisation and may require modification during in testing in order to provide clearer outputs.
- **Distribution of Work:** The streaming engine must allow for configuration of the workload across components, threads, virtual machines and nodes. This requirement will enable testing of the scaling capabilities of the solution.

3.2.8 Information Sources

Our framework must be able to utilise external information for both enrichment purposes and for scoring purposes. The external information can be static (e.g. known hosts) or dynamic (e.g. threat intelligence). The following sources must be supported as part of the POC:

- **Static Lists:** Data stored in static lists either loaded from the file system or drawn from a database. This includes geolocation data for the source and destination IP

addresses. For optimal processing latency this information must be sourced locally and not from a remote API.

- **Dynamically Updating Information:** The system must demonstrate the use of dynamic information feeds such as a threat intelligence source for scoring purposes. As with the static lists, the information should be sourced locally and not looked up remotely.

3.2.9 Operational Data Store

An operational data store is required for caching or temporary storage of transient data that may be used by components. This is ideally an in-memory key value storage that can be accessed by multiple components simultaneously with low latency.

3.2.10 Analysis and Reporting

The final requirement for Themis is handling of the output from the scoring and tagging. This has three general requirements:

- **Visualisation:** A means of visualising the scored flows using UI controls such as line charts, bar charts and maps is required. In addition, the selected tool must be able to display the data as it is generated.
- **Alerts/Notifications:** We need to be able to raise alerts or notifications when conditions are matched on scored flows (e.g. unusually high scores, large number of specified tags, etc).
- **Trends and patterns:** We need to be able to analyse and identify the long term trends and patterns in our scored flows. This information can be fed back into the scoring framework in order to detect anomalies or deviations from normal behaviour.

3.3 Summary

In this chapter we have laid down the framework requirements in terms of data enrichment, scoring, tagging, data outputs and performance criteria. Taking all of this into account,

we have specified an architecture that will meet our requirements. This architecture is described in stages, from a high level conceptual processing workflow to a more detailed design and finally, we specified a stream processing pipeline. The final section of the chapter lays down more detailed requirements for Themis which serve as a guide for the implementation described in the following chapter.

preprint

Chapter 4

Implementation

This chapter documents the implementation of Themis described in the previous chapter. During the implementation process decisions were made that may not have corresponded with the original design but were required for practical reasons. These changes are documented along with the reasoning for the changes.

This chapter starts with a Technology and Tools section detailing the applications, languages and components which were used to build the scoring topology and supporting infrastructure (section 4.1). In sections 4.2 and 4.3 respectively the flow ingestion and data sources are described. The different types of bolts and their implementation are described in sections that logically group the functionality into enrichment, scoring, utility and outputs (sections 4.4 to 4.7). The scoring topology itself is presented in section 4.8 and in section 4.9 the output analysis is discussed. Where applicable, the website from which software was sourced is provided in footnotes.

4.1 Technology and Tool Discussion

The different technologies and tools used in the framework are illustrated in figure 4.1 and briefly discussed in this section, while some background is provided regarding why they have been chosen for the project.

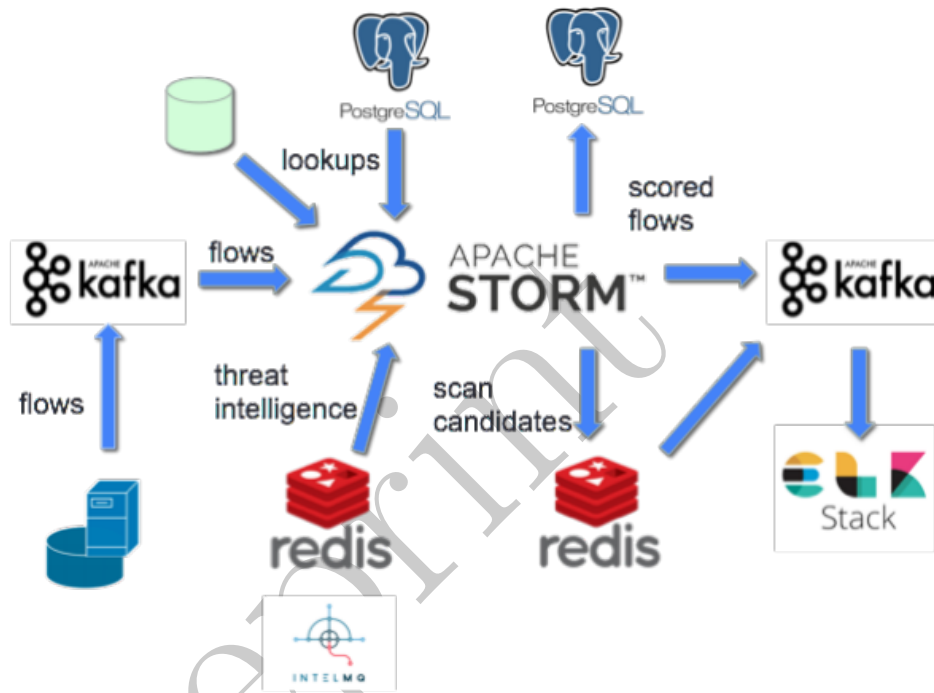


Figure 4.1: Themis technology stack

4.1.1 Queuing Frameworks

Queues serve to decouple data producing applications from data consuming applications. The ecosystem in which Themis will run consists of different components, each of which has different types of functionality (eg. NetFlow collection, scoring, visualisation, etc). Using queues between these components makes for a clean, natural design wherein each component can be implemented, deployed and tested independently of the other. Key requirements for queuing in Themis are reliability, the ability to handle volumes and low latency. With this in mind, Kafka was chosen as our queuing platform.

Kafka¹ has common queueing functionality very similar any other Publish/Subscribe broker (such as RabbitMQ², Apache Qpid³, etc...) including familiar concepts such as producers, consumers, topics, subscriptions, etc. What makes it different is the features that allow it to function at scale. Data is persisted to disk in immutable transaction logs which can be partitioned by topic across multiple brokers in a cluster. Consumers read

¹<https://kafka.apache.org/>

²<https://www.rabbitmq.com/>

³<https://qpid.apache.org/>

from a single log, eliminating the need for a queue per consumer (and consequently no duplication of data). The log partitions can be optionally replicated across multiple nodes through configuration, thereby providing fault tolerance and reliability. Kafka has also been designed and implemented with reliable, low latency, streaming in mind and offers exceptional performance in this regard (Kreps, 2014).

4.1.2 Event Stream Processing Systems

There are a number of open source event stream processing engines available. Before deciding on one, a review was undertaken of some of the more well-known big data options. In the end, Apache Storm was chosen, but it is worth considering briefly a few of the other options.

Hadoop When considering big data processing technologies, Hadoop is usually the first tool that springs to mind. It is not a stream processing platform, but for completeness, we will discuss its applicability to our use case. It has been around for 12 years and is often the first choice for processing large data sets. As such, it's a good solution for large, one-pass computations, however, much less efficient for multi-pass processing requirements. The problem stems from the MapReduce processing model, which is essentially a batch processing model. Each step in a data processing workflow has a single Map phase and a single Reduce phase. This requires reducing computations to a series of MapReduce steps in order to process your data. While each step may be efficient, the output from each step has to be stored on the distributed file system before the next step can proceed. The replication and disk IO overheads inherent in this process slow down the entire process, regardless of how efficient each job may be. This is clearly not a solution that meets our design requirements of high throughput, low latency flow scoring (see section 1.1).

Apache Spark Spark is a cluster computing platform design for fast, scalable computations. Spark makes use of Hadoop for processing and storage purposes, but it extends the MapReduce model and implements its own cluster management and computation engine. Application workloads include batch processing, iterative processing, interactive processing and stream processing. Stream processing is implemented through micro-batching, however, which introduces latency to the processing time but with the benefit of stateful, exactly once computation. The richness of the

framework and the variety of computation operations make it a good candidate for our solution. In the end, however, the complexity and the extra latency overhead counted against it in favour of the more simple Storm platform (discussed in more detail later in this section).

Kafka Streams A fairly new addition to Kafka is a library for streams processing. This enables the construction of processing topologies which consist of Kafka topics connected via processing operations. This option is very limited in its capabilities and being a library rather than a framework, it lacks many of the features that other streaming options offer. After consideration, it was determined that Kafka Streams could work as a complementary option for performing certain types of tasks such as aggregation or summation functions, but it was not suited for the primary task at hand.

Complex Event Processing (CEP) While not strictly speaking event stream processing tools, complex event processing frameworks require a mention (e.g. Drools Fusion⁴, Esper⁵, etc). These have been around for longer than the stream processing options and offer a subset of the features. A major difference is that in stream processing engines a processing graph of operators is implemented using either built in functions or custom logic. Events are streamed into this processing graph and the operators process the events before sending them onto other operators. The engines provide support for distributing this processing graph across many nodes in parallel. In contrast, CEP engines process streams using high level languages (e.g. SQL) and typically run on a single node.

There are other streaming frameworks such as Flink or Samza that could have been used. It was, however, decided to only investigate a few of the more well-known and widely used options as candidates.

The decision to choose Apache Storm as our stream processing engine was taken based on the following features:

- Scalability: Storm has been built to efficiently scale, transparently spreading data processing tasks across nodes, threads and processes as required.

⁴<https://www.drools.org/>

⁵<http://www.espertech.com/products>

- **Reliability:** Storm implements guaranteed at least once message delivery ensuring that no message is lost.
- **Fault tolerance.** When deployed in cluster mode across multiple machines, Storm can detect node failures and reassign processing tasks as needed.
- **Ease of Implementation:** Storm is programming language agnostic: bolts and spouts (see definitions below) can be defined in any language using the simple Storm communication protocol (Clojure and Java are supported natively). Storm bolts are relatively simple to develop using a common implementation pattern. Topologies are defined using a simple fluent⁶ interface.
- **Performance:** Storm has proven performance in terms of throughput and latency. For example, Cisco have deployed a solution that processes 1.2 million records per second in real time (Sirota and Dolas, 2014).

In order to understand the implementation of the scoring framework, we need to discuss the core Storm concepts and building blocks. The first concept is that of a tuple, the primary data structure used in Storm. A tuple is a collection of key/value pairs that encapsulates a unit of data streamed through the engine. The values can be of any type and the fields are dynamically typed - no declaration required (see code listing C.1 for an example of what can be put into a tuple). In Themis, the entire flow record is passed around as a single, serialised entry in a tuple.

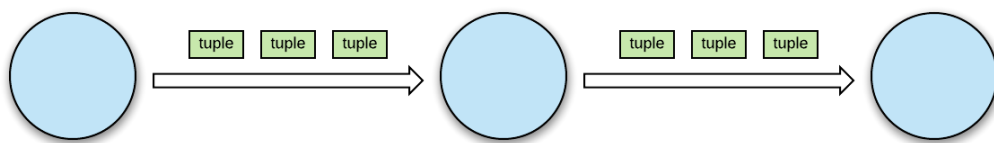


Figure 4.2: Stream of tuples

The core concept is that of a stream, a stream is an unbounded sequence of tuples that flows between nodes in the Storm engine as illustrated in figure 4.2.

The nodes make up the working primitives in Storm that perform operations on the tuples in the stream. There are two types of nodes: spouts and bolts. Spouts are special

⁶<https://martinfowler.com/bliki/FluentInterface.html>

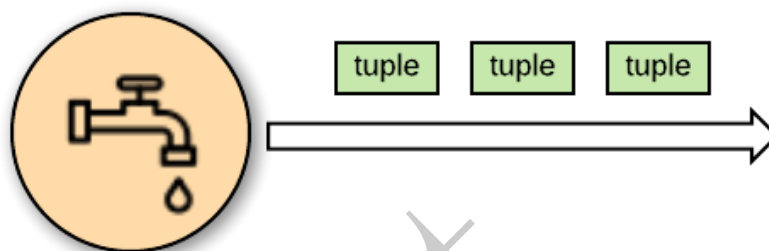


Figure 4.3: Tuple spout

types of nodes that are the source of tuple streams. Spouts are typically represented in Storm using a tap symbol as shown in figure 4.3. The spout can create the tuples programmatically, or more commonly, these are ingested from an external data sources such as queues, log files, etc. This data is then generated as a tuple stream:

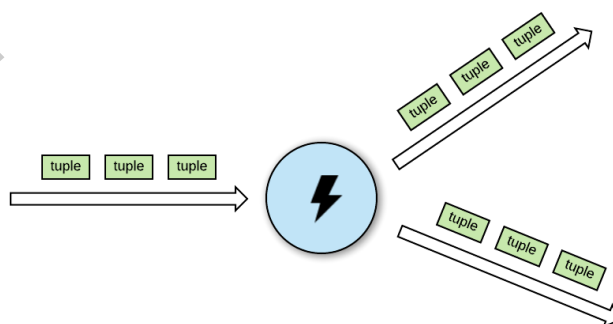


Figure 4.4: Tuple bolt

Bolts are the other type of node and are responsible for processing the stream one tuple at a time through transformations, computations, external API calls, outputting, etc. Bolts accept tuples from one or more input streams and then create one or more new output streams based on the implementation logic. In Storm a bolt is represented using a lightening symbol as shown in figure 4.4. Typically each bolt performs one and only one action on tuples.

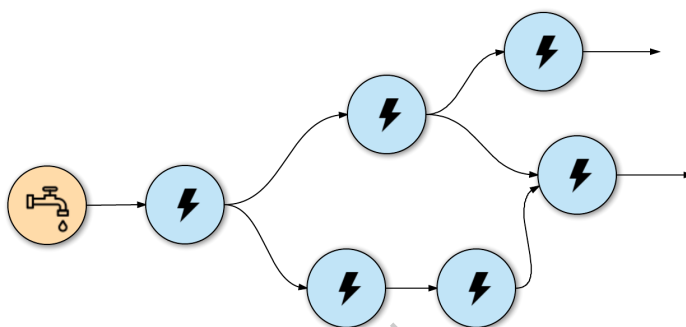


Figure 4.5: Topology of bolts and spouts

Multiple spouts and bolts are connected together into a topology (see figure 4.5). This consists of a directed acyclic graph of nodes (spouts and bolts) connected by streams. This topology is constructed, deployed and run inside a Storm instance running on a single node, or across multiple nodes.

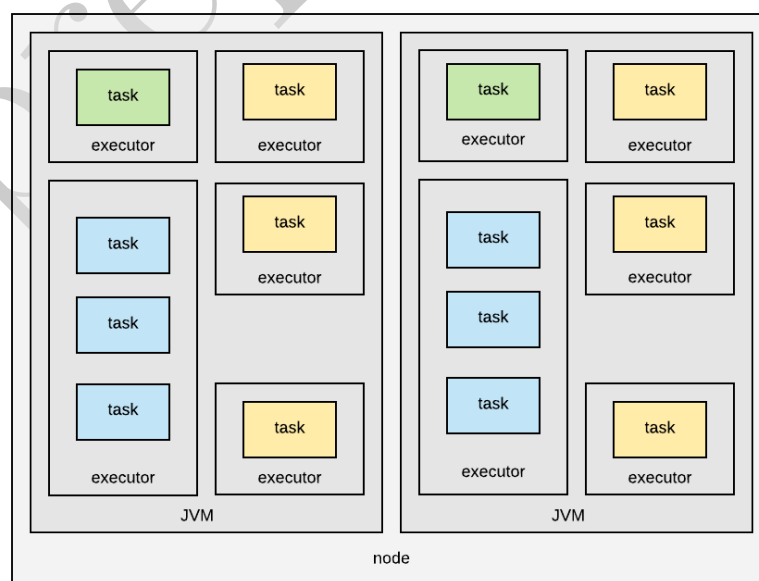
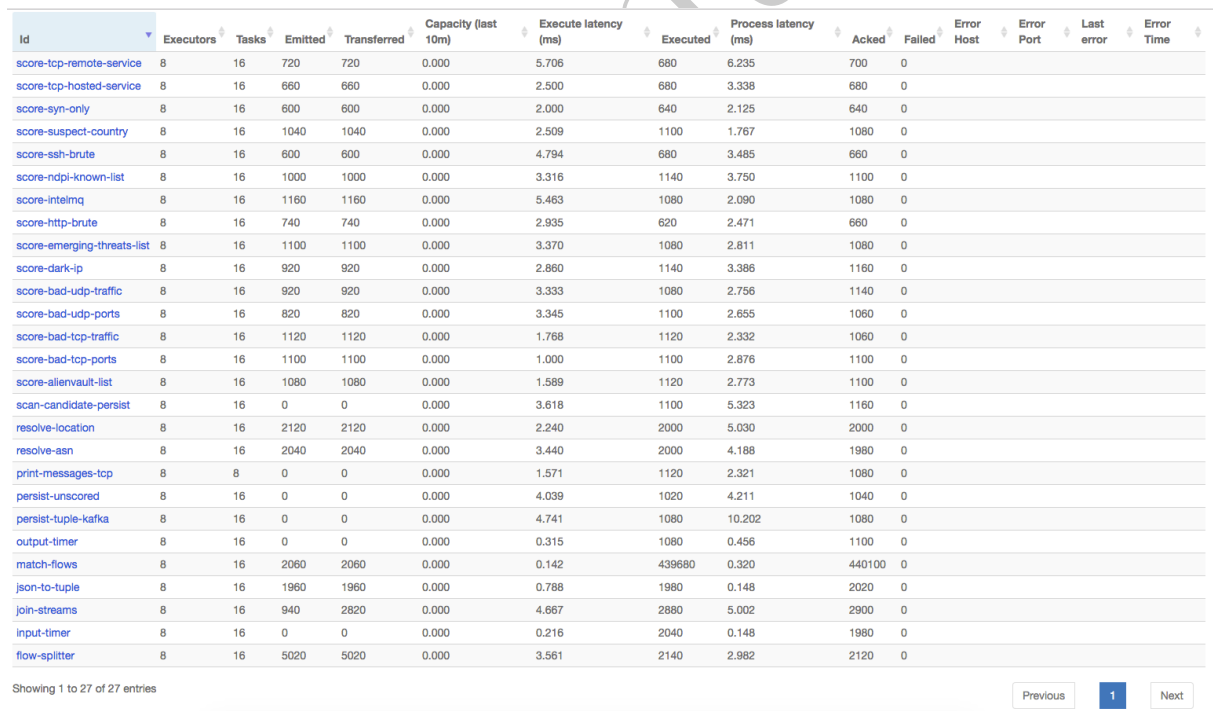


Figure 4.6: Topology scaling concepts

Scaling and concurrency are managed through the parallelism configuration of Storm. To understand how Storm scales, we need to consider the different concepts that make up a deployment as illustrated in figure 4.6. The first concept to consider is nodes. Nodes are the physical machines on which the deployment runs. A Storm cluster consists of one or more nodes that can easily be scaled up or down as required. Storm takes care of spreading

load and data transparently across the cluster. Within each node there are one or more workers. Workers represent an instance of a JVM running a storm topology. Again, these can be scaled up or down as required. Within workers a number of executors are defined. The number of executors are the number of threads running within the JVM dedicated to running the topology. The final configuration item is the task instance configuration. Each spout or bolt can be individually configured as to how many instances (tasks) of each should be created. Achieving optimal performance may require adjusting various combinations of these parameters in order to find the best combination.



Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
score-tcp-remote-service	8	16	720	720	0.000	5.706	680	6.235	700	0				
score-tcp-hosted-service	8	16	660	660	0.000	2.500	680	3.338	680	0				
score-syn-only	8	16	600	600	0.000	2.000	640	2.125	640	0				
score-suspect-country	8	16	1040	1040	0.000	2.509	1100	1.767	1080	0				
score-ssh-brute	8	16	600	600	0.000	4.794	680	3.485	660	0				
score-ndpi-known-list	8	16	1000	1000	0.000	3.316	1140	3.750	1100	0				
score-intelmq	8	16	1160	1160	0.000	5.463	1080	2.090	1080	0				
score-http-brute	8	16	740	740	0.000	2.935	620	2.471	660	0				
score-emerging-threats-list	8	16	1100	1100	0.000	3.370	1080	2.811	1080	0				
score-dark-ip	8	16	920	920	0.000	2.860	1140	3.386	1160	0				
score-bad-udp-traffic	8	16	920	920	0.000	3.333	1080	2.756	1140	0				
score-bad-udp-ports	8	16	820	820	0.000	3.345	1100	2.655	1060	0				
score-bad-tcp-traffic	8	16	1120	1120	0.000	1.768	1120	2.332	1060	0				
score-bad-tcp-ports	8	16	1100	1100	0.000	1.000	1100	2.876	1100	0				
score-allenvault-list	8	16	1080	1080	0.000	1.589	1120	2.773	1100	0				
scan-candidate-persist	8	16	0	0	0.000	3.618	1100	5.323	1160	0				
resolve-location	8	16	2120	2120	0.000	2.240	2000	5.030	2000	0				
resolve-asn	8	16	2040	2040	0.000	3.440	2000	4.188	1980	0				
print-messages-tcp	8	8	0	0	0.000	1.571	1120	2.321	1080	0				
persist-unscored	8	16	0	0	0.000	4.039	1020	4.211	1040	0				
persist-tuple-kafka	8	16	0	0	0.000	4.741	1080	10.202	1080	0				
output-timer	8	16	0	0	0.000	0.315	1080	0.456	1100	0				
match-flows	8	16	2060	2060	0.000	0.142	439680	0.320	440100	0				
json-to-tuple	8	16	1960	1960	0.000	0.788	1980	0.148	2020	0				
join-streams	8	16	940	2820	0.000	4.667	2880	5.002	2900	0				
input-timer	8	16	0	0	0.000	0.216	2040	0.148	1980	0				
flow-splitter	8	16	5020	5020	0.000	3.561	2140	2.982	2120	0				

Showing 1 to 27 of 27 entries

Previous 1 Next

Figure 4.7: Storm management UI

Storm provides a management user interface (see figure 4.7) that includes per component statistics in order to assist with monitoring and tuning. This provides insight into statistics such as the number of tuples processed per component, individual component latency, end-to-end latency and component capacity.

4.1.3 Operational Data Store

There was a requirement in the implementation for high-speed, temporary, shared data storage for caching and lookup purposes. Initially, Apache Cassandra⁷ was considered due to its widespread use in big data implementations. It was, however, found to be overkill for what was needed and the overheads associated with its operations and programming model became more of a hindrance. The amount of effort required to understand the configuration and data modelling approach, while not significant, would have taken more time than could be spared. After considering other options and their pros and cons (Kovacs, 2014), it was decided that the requirements were much more easily met by Redis⁸. The framework makes use of the high-speed key-value storage for caching and lookups, and the counter data structures for tracking statistics. Installation and setup was trivial using the Linux package manager *apt-get*.

4.1.4 Persistence

The PostgreSQL RDBMS database server⁹ was chosen for storing configuration and lookup data, as well as for the persistence of stored flows. This choice was made based on the rich feature set and well-known performance characteristics of the database server.

4.1.5 Languages

Two programming languages were used for the implementation of Themis. For the actual bolts in the Storm topology, Java was chosen due to its tight coupling to the engine (Java bolts are run natively in the engine). For all other implementation work, Python was chosen for its utility and speed of development features. In appendix H details are given on how to obtain a copy of the source code.

4.2 Implementation Overview

In this section, a high-level overview of the way in which flows are ingested, processed and outputted is presented. As discussed, in chapter 3 the focus of the work is on the scoring

⁷<http://cassandra.apache.org/>

⁸<https://redis.io/>

⁹<https://www.postgresql.org/>

and tagging framework. How the data is sourced and what is done with it post-processing is less of a concern in this work, but is addressed for purposes of completeness.

4.2.1 Environmental Context

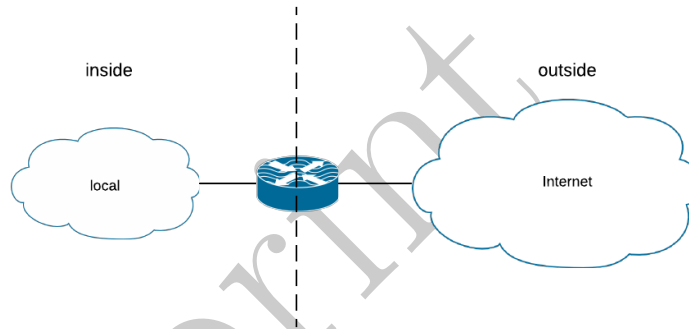


Figure 4.8: Inside and outside Network

Before discussing flow processing, we need to consider the context of the network flows, as this is an important requirement for processing and scoring the flows. The environmental context consists of a number of different facets which are dependent on the the environment in which the NetFlow data was captured. Firstly, we need to consider flow direction. NetFlow data is typically collected on a border router which routes between an organisational network (the inside) and the rest of the Internet (the outside) as shown in figure 4.8.

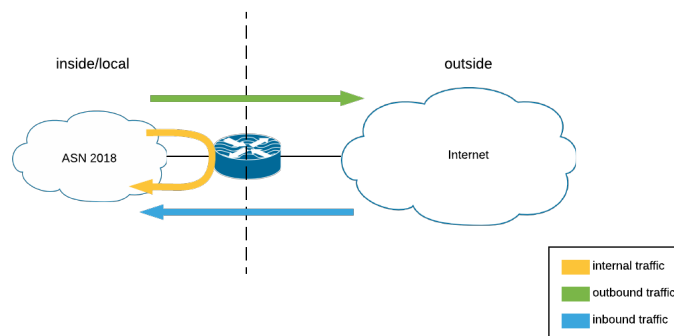


Figure 4.9: Traffic categories

In order to identify the flow direction, we either need to know the IP address ranges on the inside network or we need to know the ASN of the internal network, if applicable.

This allows us to then categorise flows into one of three different classes as illustrated in figure 4.9. Incoming traffic is network conversations initiated from the outside, outgoing traffic is conversations initiated from the inside and internal traffic is traffic between hosts on the inside which may be picked up by NetFlow. The last category is not analysed by Themis.

The next facet to consider is the IP address space in use on the internal network. This is important for two reasons: firstly we need to know what hosts are on the network in order to assist in classifying traffic to known hosts, secondly the monitoring of traffic to unallocated (or dark) IP addresses assists in detecting threat behaviours such as DDoS attacks, network reconnaissance and network scans (similar in operation to a network telescope, see Moore *et al.*, 2004).

Finally, the type of server and service hosted on the known IP addresses affects the context within which we will score traffic. While it may not be practical to document all servers and services, this information can prove valuable in terms of scoring known or expected traffic as good in order to filter it out of the final analysis and unknown or unexpected traffic as bad and this candidate for investigation. For example, we would expect to see SMTP traffic from a known mail server, but not necessarily FTP traffic.

In our implementation, the environmental context information is required as part of the startup or configuration information and is loaded from a number of different data sources.

4.2.2 Flow Scoring Process

Flows records are streamed as tuples through the Storm processing engine and scored and tagged as they transit. The flow records are extracted from the tuples into Java objects in the bolts, processed and then forwarded on again as tuples. In these objects we track the current score and corresponding tags of a flow.

The flow record contains a total for the good and bad score, as well as corresponding collections holding records for each of the individual good or bad scores assigned to the flow. These are stored in Java Sets on the flow record and allow for the analysis of scored

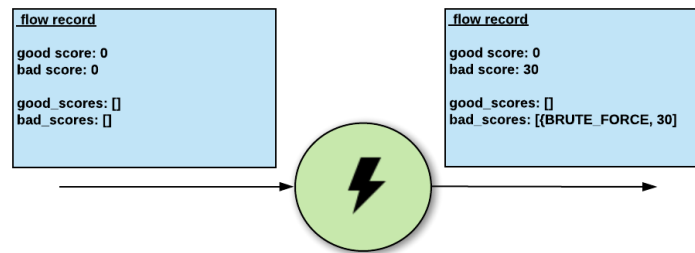


Figure 4.10: Bolt scoring

flows in terms of different scoring approaches. In the example in the diagram (figure 4.10), we see how the scoring bolt has increased the bad score by 30 and added an entry for this into the bad_score collection.

As part of the implementation a category attribute was added to the scoring and tagging process. This was to allow for categorisation of types of scoring results for analysis and visualisation purposes. Examples of categories are scanning, brute force attempts, threat intelligence sources, etc.

4.2.3 Flow Processing Overview

The process of scoring starts with the ingestion of NetFlow data by Themis from collectors. The capture, export and collection of NetFlow data is well understood and is not part of the scope of this work. However, we will discuss conceptually how the flows are sourced for completeness. NetFlow data can be collected on a network in multiple locations from multiple devices. The resulting flow data is exported to one or more centralised collectors (see section 2.1.1 and figure 2.2 for more detail).

From the NetFlow collectors the flow data is then submitted into Themis for processing via Kafka where the records are queued for processing. The Kafka queue is the entry point into the framework and allows for multiple sources of flow data to be ingested for processing. The Kafka queue aggregates the flow data from these sources and serves as a single stream of flow data for the processing framework (figure 4.11).

The core of Themis is an Apache Storm instance running our flow score topology. As discussed, a topology consists of a directed acyclic graph of bolts (see figure 4.12),

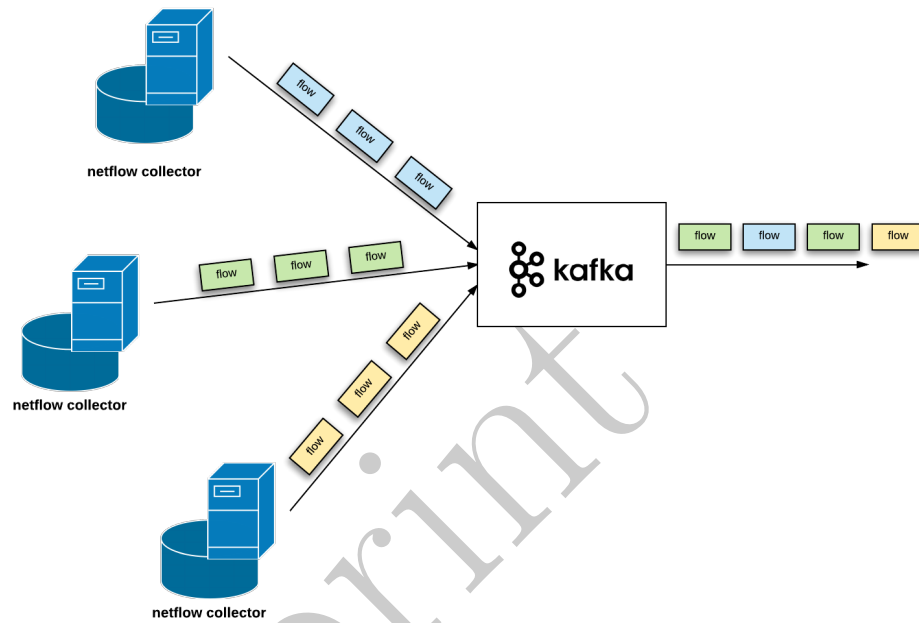


Figure 4.11: Flow ingestion

which are the basic components in which the tuples of the event stream are processed. In Themis, we have four types of bolts: enrichment, utility, output and scoring. The enrichment bolts add data to the flows, utility bolts aid in the stream processing and output bolts are responsible for persisting scored flows to external data storage or queues. These bolts are assembled in a Storm topology that encapsulates our processing and scoring workflow.

The scoring bolts are responsible for evaluating flow records and, if deemed necessary, updating the scores and tags as explained in section 4.2.1. Each bolt tests one, and only one, condition and adds a score and tag to the flow tuple if that condition is met. Meeting a condition can result in the addition of a positive score or a negative score for the flow. The scores and tags are cumulative with new ones added to the existing scores and tag list as required. Figure 4.13 illustrates this concept by showing a tuple progressively moving between a subset of scoring bolts and being scored by two out of three of them. With each successful scoring, the tuple score and tags change.

The flow records are ingested into the Apache Storm instance by a Kafka spout that sources the flow records from the Kafka queue and streams them into the scoring topology.

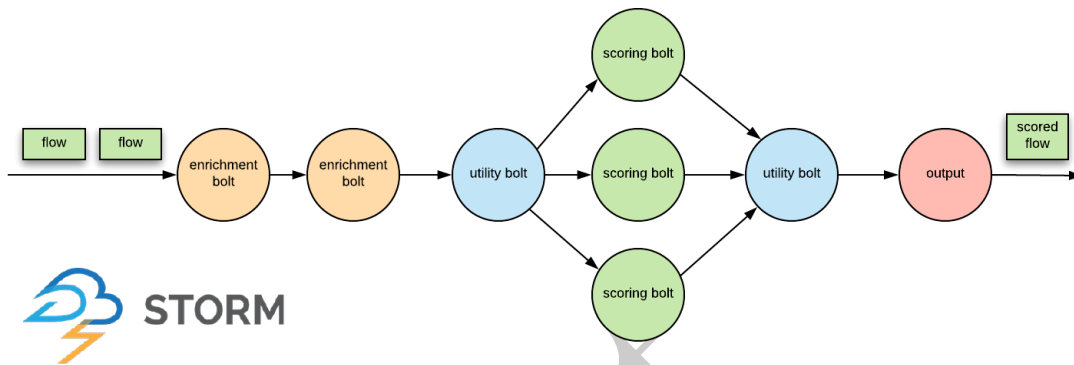


Figure 4.12: Flow scoring workflow

In order to enrich the flows and aid with the scoring, the Storm topology needs access to configuration information, enrichment data and threat intelligence from multiple external data sources. As illustrated in the example in figure 4.14, bolts may access this data from different sources (Postgres and Redis are the two sources in the diagram).

In order to keep our goal of low latency and high throughput, these data sources need to be readily available for use by the bolts in the topology. To meet these objectives the information is either loaded into memory on startup (ideal for static data) or is made available via low-latency persistence data store (ideal for data feeds). In this implementation of the framework, a number of approaches have been taken depending on the nature of the data (static vs dynamic) and the size of the data set. The final step in the flow processing is the output of the processed flow record for use in upstream systems (figure 4.15).

In Themis, the following three categories of output have been implemented:

- **Log File:** The Themis implementation makes use of the log4J libraries for logging debug information. The logging library has been used for the implementation of an output bolt to log the scored flow record to file.
- **Database:** A bolt has been implemented that will log scored flow data to a database. This data can be used for traditional batch processing analysis, archival purposes, further processing, temporal threat intelligence, etc.

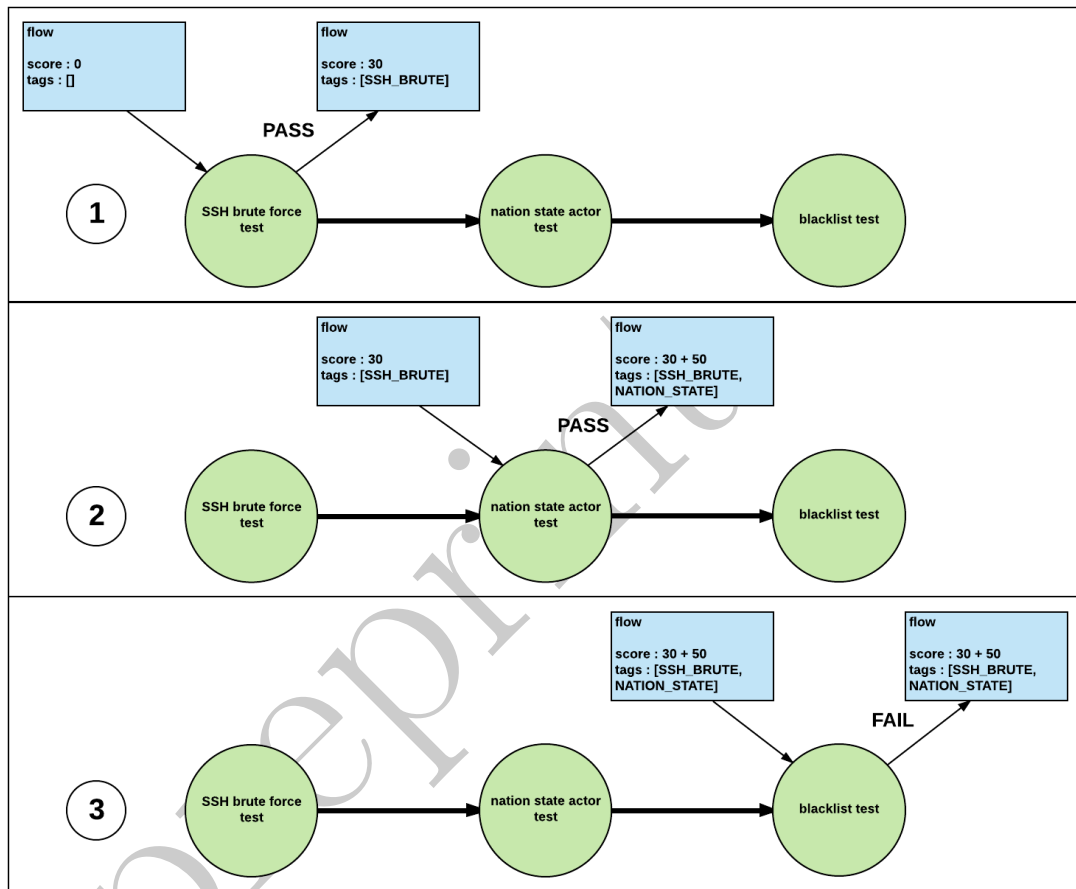


Figure 4.13: Additive scoring example

- **Kafka:** A bolt has been implemented that allows for the queuing of scored flows to a designated Kafka topic. The queue can be monitored by event management software for alerts and notifications, or by visualisation tools for dashboards and trend analysis.

For the purposes of this work, the data queued to Kafka is further ingested into an ELK stack (Elasticsearch, Logstash, Kibana) and a number of dashboards have been created to visualise the scored flows. Additionally, statistics have been drawn from the scored flow data persisted to the database.

4.3 Flow Ingestion

The first step in the processing is the ingestion of flow data. In order to make Themis open to multiple sources of NetFlow data, the ingestion is done by submitting flow data to

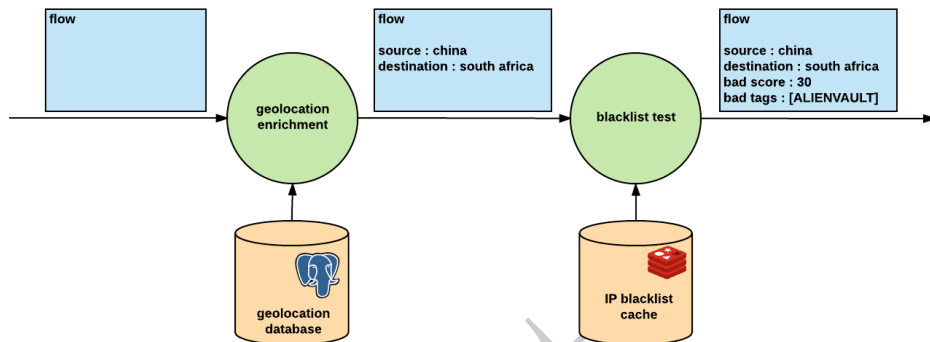


Figure 4.14: Scoring from external data sources

a topic in a Kafka instance (netflow-json). For consistency and simplicity, we require that the flows are encoded in JSON before ingestion. For our implementation, a python script was implemented that converted flow data records into JSON and sent this to Kafka. The JSON schema in listing B.1 in appendix B specifies the required data fields.

In listing B.2 an example of a JSON encoded flow can be found. Note that IP addresses from the internal network have been anonymised during ingestion using addresses described in RFC5737 (Arkko *et al.*, 2010).

For the purposes of this project, the input interface boundary is the Kafka queue where flow records are ingested. For development and testing purposes, a sample of NetFlow data was loaded into a database table for ease of access. This table contains only those fields required for scoring (see listing A.1 in appendix A).

This data was originally collected using the nfdump tools and stored in native nfcapd format in files containing five minutes of NetFlow data each. The flow data was extracted into CSV format using the nfdump tool and bulk loaded into the database. To simulate flow ingestion, a python script was implemented that loads data from the database, converts it to the prescribed JSON format and submits it to the Kafka queue for processing. The python script was implemented with the following features:

- Number of Rows: The user can specify how many rows to submit to the database. This option is useful when testing new functionality with a small number of rows at a time. Flow records are submitted sequentially in order of the flow timestamp.
- Starting Flow Time: The user can specify the timestamp from which to start submitting. This is used to continue from where the last run left off.

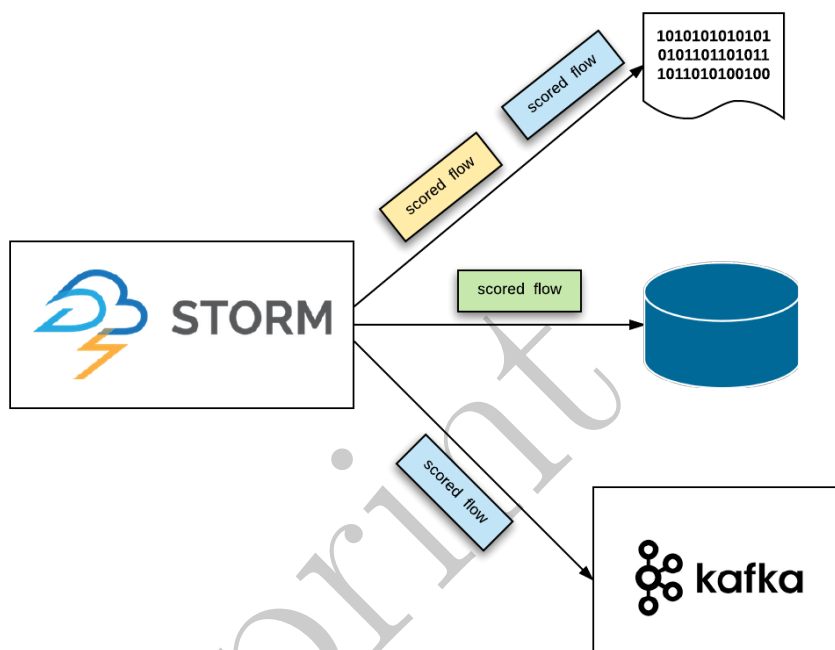


Figure 4.15: Output of scored flows

- Submission Mode: Flow records can be submitted in one of two modes - as fast as possible and timestamp regulated. In the first option, the flows are queued in Kafka as fast as they are returned from the database. In the second option, the script will regulate the flow into the queue to match the collection time represented in the flow timestamp. In addition, this mode will change the flow timestamp to the current time to emulate real-time collection of data.

In a production deployment, a modified version of this script could be used with the NetFlow collector and exporter to directly ingest NetFlow records as they are sourced from the network.

4.4 Data Sources

In the flow processing overview, mention was made of using data from third party sources for configuration and scoring purposes. In this section, we cover these data sources and the implementation that makes them accessible in Themis. The list of sources and their usage is not intended as a complete list, but rather to demonstrate the many different ways that information can be configured or made available in the framework. In a lot of

cases, the information is loaded at runtime which is a limitation in terms of flexibility. This shortcoming can be easily overcome through either polling for new data periodically or through some form of signalling whereby Themis will reload configuration on request.

4.4.1 Internal Network Configuration

In order to differentiate the internal network from external network, it was determined that for some organisations it is possible to use the Autonomous System Number (ASN) of the internal netblock as a means of identification. To use this information the ASN numbers in question are loaded from a simple text file on startup and stored in memory. This in memory storage is encapsulated in a class that exposes a boolean method that tests to see if a supplied ASN is in the list. This allows implementation logic in Themis to easily determine where a flow source or destination IP address belongs. A list of the known internal IP addresses is also loaded into memory from PostgreSQL on startup.

4.4.2 IP Address to ASN lookup

For both enrichment and flow direction purposes we need to identify the autonomous system to which an IP address belongs. The definitive source for this information is the BGP routing tables on the Internet backbone. Querying this in real time for each flow is not, however, practical. For Themis, we have made use of the free-for-use GeoLite2 ASN database from MaxMind¹⁰. This database consists of a custom database format that is optimised for fast IP address lookups. This database is loaded on startup using the MaxMind Java libraries and then used to map IP addresses to ASNs. This provides a fast and effective way of identifying ASNs at the expense of having potentially out-of-date data. For solutions requiring more accurate information, there are APIs available that could be used in conjunction with a custom caching solution to keep lookup time to a minimum (these often come at a cost however). Another alternative would be to make use of a live BGP feed to map IPs to ASNs.

4.4.3 IP GeoLocation

IP based Geolocation information maps an IP address to a geographic location. This information can be as general as the country in which the IP is located, right down to

¹⁰<https://www.maxmind.com/en/home>

which city in that country. Enriching the flow data with this information allows for scoring based on country of origin and for analysis of the data by geographical region. It also provides insights when the data is visualised on a map. This framework looks up the Geolocation information for the source IP address and the destination IP address of each flow. The resulting values are then set on the corresponding country, country code, city and longitude/latitude attributes in the flow tuple. If no data is found for an IP address, the location descriptors are set to “unknown” and the longitude/latitude to 0,0 (a location in the Atlantic Ocean approximately 600 kilometres south of Ghana). The lookup is done using the free-for-use GeoLite2 City database from MaxMind. This database consists of a custom database format that is optimised for fast IP address and IP netblock lookups. The framework loads the database into memory from a file disk on initialisation and makes this available for mapping the flow IP addresses to the corresponding geographic location. It should be noted that geolocation data is often not very accurate and may not necessarily give a true reflection of a host’s location.

4.4.4 IP Blacklists/Whitelists

A number of organisations provide lists of IP addresses that have been blacklisted for various reasons (CNC servers, botnets, etc.). Each source may have a different focus and/or reliability and can therefore be treated differently for scoring purposes. In addition, it is possible to get lists of IP addresses for well-known networks such as Facebook, Google and YouTube and use these as whitelists. The assumption is that traffic to and from these networks has a very low likelihood of containing malicious activity. In Themis two static lists are used for blacklist scoring (AlienVault and Emerging Threats), while one is used for whitelist scoring (using IP addresses from the openNDPI project¹¹). The data from these sources has been extracted and stored in the MaxMind IP database format. This format allows for the storing of IP addresses and netblocks in a quick-to-access format.

4.4.5 Threat Intelligence Feeds

Threat intelligence information is dynamic with new information constantly coming to the fore and as such, static lists are quickly out of date. In order to make use of dynamic feeds the IntelMQ¹² project was installed. This project enables multiple sources of information

¹¹<http://www.ntop.org/products/deep-packet-inspection/ndpi/>

¹²<https://github.com/certtools/intelmq>

to be aggregated and stored in a harmonised format. For Themis, a customised output was implemented to persist threat information updates to a Redis cache. This information was stored with expiring keys ensuring that only the latest set of information was made available. The Redis cache was then used to lookup IP addresses for matching threat intelligence information.

4.4.6 Country Watch List

A list of countries required for scoring is stored in the PostgreSQL database and loaded on startup. This is stored as a list of country codes that can be checked against for matches using geolocation data added to the flow tuples. In addition, each country code has a weighting that can be used to alter any scores associated with a match.

4.4.7 TCP/UDP Services

The final type of lookup is a list of TCP or UDP ports that are used for checking for approved or unapproved traffic flows. The information is stored in the PostgreSQL database as a IP address/Port number/Protocol combination. In addition, a flag is stored on each record indicating whether or not the combination represents a service hosted on the IP address or a possible remote service that would be accessed by the host in question. This structure allows for the definition of known services on hosts (e.g. this host is a web server). It also allows for the definition of expected traffic for non-hosted services (e.g. this host is an outgoing SMTP server and should be sending traffic to remote hosts on port 25). The configuration information is used to determine if observed traffic is unusual or not.

4.5 Flow Enrichment Bolts

Enrichment bolts add meta data to the flow records as they are processed. This enrichment can be used either in the scoring or later on in the analysis of the scored flow output.

4.5.1 JSONtoTupleBolt

The first bolt in our topology must be this bolt. Its function is the conversion of the JSON formatted NetFlow records to an internal Java data transfer object (DTO). The reasons for the conversion are:

- To use a consistent internal format for storing the record that is independent of any external serialisations. If at some point we change the format of the ingested NetFlow data, then only this bolt would need to be updated or replaced in the topology.
- Since our bolts are implemented using Java, accessing and manipulating a native Java object will be more efficient for performance. Should there be a requirement to process tuples in bolts implemented in a different language, the native Java class can be converted to JSON, BSON or another such similar cross-platform format.

The Java DTO used to store the flow records is transmitted through the topology as Storm tuples. The DTO stores the original NetFlow data, enrichment information, control data, the flow scores and a collection of tag information. The DTO is implemented in the `NetFlow.java` class as shown in listing C.1 in appendix C.

The good and bad scores, and associated tags are stored in the DTO as a set of `FlowScore` objects. Each entry in the set contains an individual score along with a category and a code (both of which make up the score tag). The implementation of which is documented in listing C.2 in appendix C.

Other than getters and setters, the only significant method on the `FlowScore` object is one to calculate the total scores. This method iterates through the set of `FlowScore` objects and sums up the individual good and bad scores, and then sets this on the `FlowScore` object. This method is provided as a means of totalling the scores before persisting or outputting the final scored flow.

4.5.2 ASNBolt

An Autonomous System Number (ASN) is an identifier used in BGP routing to group networks belonging to the same organisation (system). This attribute is then used when sharing routing information between these organisations. For threat intelligence purposes, the ASN can be used to assist in identifying the source organisation of an IP address.

The ASN bolt looks up the Autonomous System Number to which the source IP address and the destination IP address of the flow belong. The resulting values are then set on the `src.as` and `dst.as` attributes in the flow record. If a value cannot be found for an IP address, then the corresponding attribute is set to 64496. This number is taken from RFC 5398 - ASN numbers reserved for documentation (Huston, 2008). In reporting and analysis, we can then take into account any IP addresses for which no ASN was identified. The lookup is done using the free-for-use GeoLite2 ASN database from MaxMind as described in section 4.4.2.

4.5.3 GeoLocatorBolt

IP based Geolocation information maps an IP address to a geographic location. This information can be as general as which country the IP is located in down to which city in that country. Enriching the flow data with this information allows for scoring based on country of origin, allows for analysis of the data by geographical region and provides insights when the data is visualised on a map.

This bolt looks up the Geolocation information for the source IP address and the destination IP address of the flow. The resulting values are then set on the corresponding country, country code, city and longitude/latitude attributes. The lookup is done using the free-for-use GeoLite2 City database from MaxMind as described in 4.4.3.

4.6 Flow Scoring

As discussed previously, Storm performs stream processing by routing streams of records (or tuples) through a topology of nodes (or bolts). Each of these bolts performs distinct operations on the tuples before passing them onto the next node or nodes in the topology until the processed flow data is finally emitted. In this section we describe the bolts that have been implemented in order to support the flow scoring and tagging.

4.6.1 ScoreBolt

This bolt extends the Storm *BaseBasicBolt* class and forms the base class for all the scoring bolts implemented in the topology. All Java classes implemented for scoring

purposes must extend it. Its purpose is to provide a common set of attributes across all scoring bolts, score value, score code and score category (the last two are used for the tagging). These values are used in the scoring bolts when assigning scores and tags to matching flows and are set at compile time when the Storm topology is defined.

In any extended class the base class constructor must be called before any other initialisation work is done. In all cases, scoring bolts must include - as part of their initialisation a score category - a score tag and a score value. These values are then appended to the flow tuple, if the executing logic deems it appropriate. This is represented by the pseudo code in listing 1 in the scoring bolt documentation throughout this section.

apply the tag and weighted score

Algorithm 1: Scoring and Tagging Example

4.6.2 ScoreCountryBolt

The purpose of this bolt is to negatively score flows where either side of the traffic originated in a defined country. In addition, the configuration allows for different weightings for different countries. This allows, for example, traffic from both China and Russia to be scored negatively, but with China traffic getting a higher negative score. The logic implemented in this Bolt is shown in algorithm listing 2.

Input: flow tuple
Output: scored flow tuple

```

if flow source ASN is our ASN then
  | if destination country is in the country watchlist then
  | | fetch the weighting for the country from in memory cache;
  | | apply the tag and weighted score;
  | end
else
  | if source country is in the country watchlist then
  | | fetch the weighting for the country from in memory cache;
  | | apply the tag and weighted score;
  | end
end

```

Algorithm 2: Country Scoring Bolt Logic

4.6.3 ScoreDarkIPBolt

Traffic directed at dark IP address space may be an indication of scanning or botnet activity. This bolt uses a list of known IP addresses on the inside in order to determine whether the traffic is directed at dark IP address space. The logic implemented in this Bolt is shown in listing 3.

Input: flow tuple
Output: scored flow tuple

```

if flow source ASN is one of our ASNs and source IP is not in our IP list then
  | apply the tag and weighted score
end
if flow destination ASN is one of our ASNs and destination IP is not in our IP list
  then
  | apply the tag and weighted score
end

```

Algorithm 3: Dark IP Scoring Bolt Logic

4.6.4 ScoreGenericIPListBolt

As part of this bolts instantiation, the details of a file are passed in containing a list of IP addresses stored in the MindMax format. This list is then used during execution as shown in code listing 4.

Input: flow tuple
Output: scored flow tuple

```

if either source IP or destination IP is in the list then
  | apply the tag and weighted score
end

```

Algorithm 4: Generic IP List Scoring Bolt Logic

In addition to the normal parameters, a boolean flag indicates whether the scoring should be applied to the goodness or badness of the flow. This generic bolt can be used for blacklist or whitelist scoring.

4.6.5 ScoreHTTPBruteForceBolt and ScoreSSHBruteForceBolt

These bolts are examples of the application of prior work to scoring in Themis. For HTTP or HTTPS brute force detection, the logic implemented follows the findings presented in Van Der Toorn *et al.* (2015). The logic implemented to check for incoming (i.e. to the internal network) connections is shown in code listing 5.

```

Input: flow tuple
Output: scored flow tuple

if flow protocol is TCP then
  Comment: We first check for HTTP traffic (port 80)
  if incoming flow is connecting to port 80 then
    Comment: Packet and byte counts as per Van Der Toorn et al. (2015)
    if (packets sent  $\geq 5$  and packets sent  $\leq 12$ ) and
      (bytes sent  $\geq 363$  and bytes sent  $\leq 1130$ ) then
      | apply the configured tag and score
    end
  end
  Comment: Then check for HTTPS traffic (port 443)
  if incoming flow is connecting to port 443 then
    Comment: Packet and byte counts as per Van Der Toorn et al. (2015)
    if (packets sent  $\geq 7$  and packets sent  $\leq 17$ ) and
      (bytes sent  $\geq 789$  and bytes sent  $\leq 2885$ ) then
      | apply the configured tag and score
    end
  end
end

```

Algorithm 5: HTTP Brute Forcing

For the SSH brute force detection, the techniques described in Hofstede and Sperotto

(2014) are used. The logic implemented in this bolt is in algorithm listing 6.

```

Input: flow tuple
Output: scored flow tuple

if flow protocol is TCP then
  if source port or destination port is 22 (SSH) then
    Comment: Packet counts as per Hofstede and Sperotto (2014)
    if packets sent  $\geq 11$  and packets sent  $\leq 51$  then
      | apply the configured tag and score
    end
  end
end

```

Algorithm 6: SSH Brute Forcing

4.6.6 ScoreInsecurePortConversationBolt and ScoreUnknown-PortConversationBolt

These are two examples of scoring flows based on port numbers. The implementations use the same base class which, in addition to the standard parameters, loads a protocol number, a list of port numbers and a minimum packet count. The two bolts then implement slight variations in logic in order to score different scenarios. The first one checks for insecure traffic by checking flow traffic ports against a known list of risky ports (code listing 7), while the second bolt checks for unknown traffic by comparing the flow traffic

to a list of known ports and scoring the exceptions (code listing 8).

```

Input: flow tuple
Output: scored flow tuple

if flow protocol matches the configured protocol then
  if source port or destination port is in the configured list of ports then
    if packets sent  $\geq$  minimum packet count then
      | apply the configured tag and score
    end
  end
end
end

```

Algorithm 7: Insecure Port Check

```

Input: flow tuple
Output: scored flow tuple

if flow protocol matches the configured protocol then
  if source port or destination port is not in the configured list of ports then
    if packets sent  $\geq$  minimum packet count then
      | apply the configured tag and score
    end
  end
end
end

```

Algorithm 8: Unknown Port Check

4.6.7 ScoreServiceBolt

This bolt is generic and can be used for good or bad traffic related to known services (e.g. for a mail server or a web server). The bolt does scoring based on either:

- **Hosted Services:** In this case, the services in question are hosted locally. This allows us to identify expected incoming traffic. For example, a known internal SMTP server or an an internal HTTP server.
- **Remote Services:** In this case, the bolt looks for outgoing traffic from a known IP on the internal network to a known external service. This would allow us to identify, for example, outgoing traffic to remote DNS servers or SMTP servers from an internal mail host.

By identifying known or expected traffic we can positively score flows that match and negatively score flows that don't. The bolt configuration includes protocol to be examined, a minimum byte count and a flag indicating whether negative scoring applies (i.e. apply score if the to or from the known host does not match the configured ports). The logic for this bolt is implemented shown in code listing 9.

```

Input: flow tuple
Output: scored flow tuple

if packets sent > minimum packet count then
  if we are looking for internally hosted services then
    if the flows internal host matches then
      if the flows internal port matches then
        | apply the configured tag and score to the goodness
      else if we are negative scoring then
        | apply the configured tag and score to the badness
      end
    end
  end
  if we are looking for externally hosted services then
    if the flows external host matches then
      if the flows external port matches then
        | apply the configured tag and score to the goodness
      else if we are negative scoring then
        | apply the configured tag and score to the badness
      end
    end
  end
end
end

```

Algorithm 9: Internally or Externally Hosted Services

4.6.8 ScorePossibleScanBolt

This bolt implements logic that looks for evidence of potential scans. The bolt does this by looking for one of the following patterns in the TCP flags attribute in the flow:

- SYN only
- FIN only (FIN probe)
- PUSH and FIN (XMAS probe)

- No flags set (Null probe)

This information alone is not enough to detect a scan and therefore this information is used in another bolt to send candidate flows out of Themis for further analysis, as scan detection requires a view of traffic flows over a period of time (see 4.7.3).

4.6.9 ScoreIntelMQBolt

The final scoring bolt in the list makes use of information gathered by the IntelMQ application. A customised output node places the discovered threat information into a Redis cache. This bolt looks up the IP addresses of flow records in the cache, and if found, will score the flows accordingly. The tag used for the scoring is taken from the IntelMQ data for purposes of clarity and traceability.

4.7 Flow Output Bolts

A number of output bolts have been implemented for persisting or forwarding the results of the processing.

4.7.1 LoggerBolt

The logger bolt can be used for outputting the current state of a flow record to file using the standard Java Log4J libraries. This logging library is used in the Storm framework, so all logging can be managed through a single interface and directed to a single location. The bolt converts the flow record tuple to JSON before outputting. The primary use for this bolt is debugging. An example of the output is shown in listing E.1 in appendix E (internal IPs have been anonymised as discussed in section 4.3).

4.7.2 PersistScoredBolt

This bolt will persist flow records to a PostgreSQL database. The bolt saves the record to database tables as illustrated in figure 4.16.

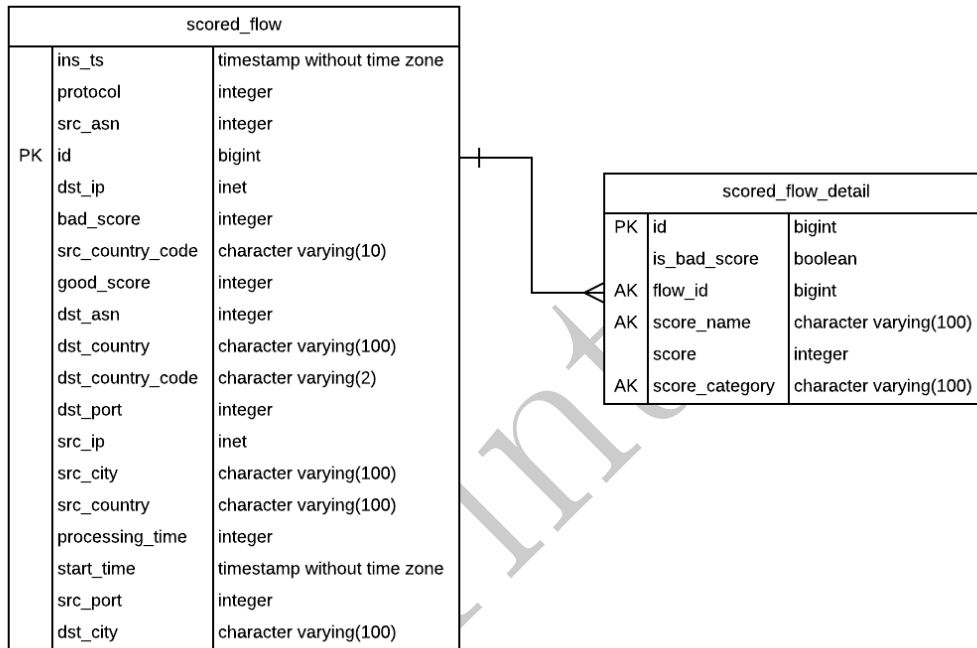


Figure 4.16: Scored flow ERD

4.7.3 PersistScanCandidateBolt

This bolt will persist information to a Redis cache to allow for scan (horizontal, vertical or distributed) detection. To do this, the bolt creates three different hashes in the cache that can then be analysed in order to perform scan detection (each of these is described in more detail later in this section). Note that the bolt does not do the actual scan detection, but instead persists flow data in a format to assist in detection. Suspect flows are flagged in the *ScorePossibleScanBolt* bolt (section 4.6.8) and then routed or streamed to this bolt and sent to Redis. A separate Python script is run periodically to look for data combinations that match scan patterns.

Vertical scan detection

The hash key is the source IP and destination IP address of a flow. In the hash the range of destination ports are tracked along with a count of bytes, packets, matching flows, first time a matching flow was seen and last time a matching flow was seen. If a vertical scan is in progress from the source to the destination IP, then the number of ports and flows in the hash will grow over time and a scan can be intimated from the traffic patterns reflected in this hash.

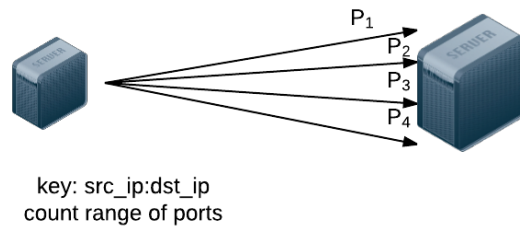


Figure 4.17: Vertical scan detection

Horizontal scan detection

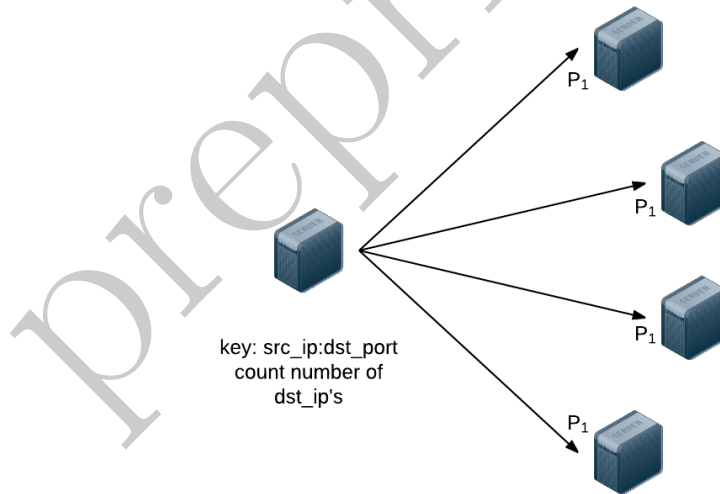


Figure 4.18: Horizontal scan detection

The hash key is the source IP and destination port of a flow. In the hash, the range of destination IP addresses are tracked along with a count of bytes, packets, matching flows, first time a matching flow was seen and last time a matching flow was seen. If a horizontal scan is in progress from the source IP to the destination port, then the number of destination IPs and flows in the hash will grow over time and a scan can be intimated from the traffic patterns reflected in this hash.

Distributed scan detection

For the last type of scan considered, the hash key is the destination port of a flow. In the hash, track is kept of all source and destination IP pairs, along with first time a matching

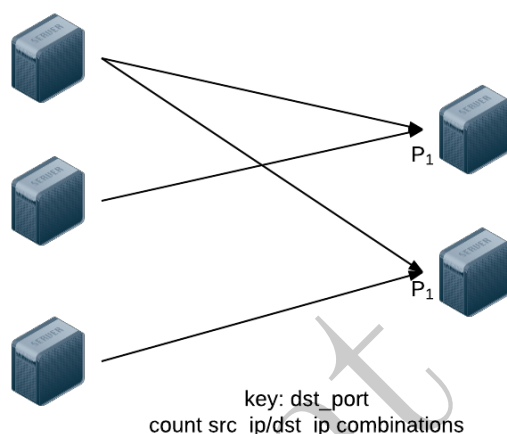


Figure 4.19: Botnet scan detection

flow was seen and last time a matching flow was seen. Over time if there is a large set of different source and destination IP pair combinations to the same port, then a possible botnet scan can be inferred.

4.7.4 PersistScoredBoltToKafka

This bolt converts the flow records to JSON and outputs the resulting text to a Kafka queue. The purpose of the bolt is to make the scored flow data available for ingestion by analysis or visualisation tools such as Elasticsearch or Kibana. The output is done as two types of records (to two different Kafka queues). The first contains the core flow data along with the score totals (see listing D.1 in Appendix D) and the second is a record per tag which includes the score for that tag and the tag category (see listing D.2 in Appendix D). Downstream systems can analyse and report on the summary scored flows or they can drill down into the detailed tags and categories.

4.7.5 PersistUnscoredBolt

The last output bolt is used to send flow data, which we are not interested in, to a Kafka queue. These flows are ones that do not match our flow scoring requirements (e.g. If we only want to look at inbound traffic, then outbound flows can be discarded but still recorded using this bolt).

4.8 Flow Utility Bolts

A number of utility bolts were implemented to assist with the processing of the data. Included in this group is a bolt that can be used for instrumenting the topology in order to gather timing and processing statistics.

4.8.1 Kafka Spout (ingestion)

A spout is a special case of a bolt that is responsible for sourcing a stream and sending the tuples into the topology. In Themis, the stream of data originates from the Kafka topic to which the raw NetFlow data has been queued as described above (the “netflow-json” queue). The Kafka spout will pull the JSON encoded messages from the Kafka queue and send them on to the next bolt or bolts configured in the topology. The spout is also responsible for keeping track of delivery of tuples. Each tuple sent is acknowledged by downstream bolts until the final bolt in the topology. The spout will track the acknowledgements and is responsible for retransmitting lost tuples or marking tuples as failed should they not make it through the topology to a terminal node. There are three spout configuration options that can be used to manage the rate at which tuples are fed into a topology:

- the maximum number of outstanding (i.e. in transit through the topology) tuples,
- a timeout for acknowledgements, and
- the number of instances of a spout (i.e. how many spouts of this type can be running at the same time).

During the implementation and testing stage of the project these options were experimented with in order to find a balance between latency/throughput and overloading the topology. The details of this can be found in the next chapter.

4.8.2 PersistTimerBolt

This is a generic bolt used for instrumentation. It can be placed anywhere in a topology and serves as a passthrough bolt whose only function is to update a preconfigured key in

the Redis cache with the current timestamp. By placing a number of these throughout the topology with different key values we are able to record and gather processing timing information. This data can then be used for throughput and latency recording and optimisation.

4.8.3 MatchFlowBolt

As has been discussed previously, NetFlow data is exported as unidirectional flows. These flows are then ingested into Themis as unidirectional flow records. To provide full context to our data, the packet and byte count of both sides of a flow is extremely useful to have. In order to get this information this bolt attempts to match the unidirectional flows that make up a network conversation and then update the byte and packet counters in each of the two associated flow records. The bolt does this by caching flow records in Redis using the source IP, source port, destination IP and destination port fields as a key. The source IP used in the key is the lowest numerical IP address in the flow. The first time a flow is observed it is serialised and stored in Redis (it is not forwarded on in the topology). When a matching flow is observed (i.e. the other side of the conversation) the first flow is removed from the cache, both flows are combined with the correlating bytes and packet counts and then a single flow record (representing a network conversation) is forwarded on through the topology. Each instance of the bolt keeps a second cache that stores flow records for a maximum of 90 seconds. If a match for a flow has not been found before this expiry period then the flow is removed from both caches and forwarded on without any additional packet or byte data. The majority of unmatched flow records will be for scans or traffic to dark IP's. This ensures that should there be missing flow records in the source NetFlow data, we still process what we have timeously.

4.8.4 FlowSplitterBolt

In order to leverage the benefits of parallelism available in Storm this bolt is used to split the stream of flow records up into a number of different streams. Each stream has certain characteristics that lend themselves to different types of scoring approaches. The following list of streams are currently implemented:

- All Flow Data: A copy of all records that we are interested in are sent down this stream.

- **Uninteresting Stream:** All records we are not going to process are sent down this stream. In our case this is all outbound flows as we are only examining inbound traffic for the purposes of this research.
- **TCP Only Stream:** Some of the scoring only makes sense for TCP traffic (e.g. detection of candidate scans using TCP flag combinations).
- **Port Processing:** This stream is used for scoring that does port based analysis and is used mainly as a means of splitting out the load of one category of scoring to its own path.
- **Batch Processing:** The scan detection output bolt described above is a terminal bolt in the system. This stream is used to direct flow data to that bolt for output to the scan processing scripts.

4.8.5 JoinBolt

The JoinBolt is a utility bolt used to aggregate two or more streams of flow data in the topology. The SplitterBolt sends a copy of each record down each stream which may result in different scores being applied to each copy. The JoinBolt is responsible for joining the streams and combining the flow copies into a single version and forwarding on. Flows are cached as they are arriving until all copies have been received. The different versions are merged and a single instance is forwarded on. The merging process consists of combining the score and tagging information from each arriving tuple with the latest version in the cache. If all outstanding tuples have arrived, the total scores are re-calculated and the final version is forwarded on. Otherwise the updated version is put back in the cache.

4.9 Flow Scoring Topology

The scoring topology is the Storm topology that consists of the various bolts combined in a DAG for stream processing purposes. The full topology can be found in figure G.1 in appendix G. The bolt names in the topology represent instances of the bolts described above. The functioning of the bolts depends on the configuration of the instances. In table 4.1 the instances of the scoring bolts and their purpose is described.

From the list above it can be seen that in some cases there is potential overlap between different scoring instances. For example, score-tcp-hosted-service and score-bad-tcp-traffic.

Table 4.1: Scoring bolts

Scoring Bolts Configuration		
Bolt Instance	Bolt	Notes
score-dark-ip	ScoreDarkIPBolt	This instance of the bolt applies negative scoring to all traffic directed at dark IP hosts on the internal network.
score-emerging-threats-list	ScoreGenericIPListBolt	This instance of a generic IP list bolt uses the Emerging Threats blacklist to score flows.
score-alienvault-list	ScoreGenericIPListBolt	This instance of a generic IP list bolt uses the Alienvault blacklist to score flows.
score-suspect-country	ScoreCountryBolt	Scoring of traffic from countries appearing in the configuration.
score-intelmq	ScoreIntelMQBolt	Any flows with IP addresses appearing in the IntelMQ sources.
score-ndpi-known-list	ScoreGenericIPListBolt	This bolt scores all traffic to hosts in a whitelist.
score-syn-only	ScorePossibleScanBolt	Negative scoring for TCP scan candidate flows.
score-ssh-brute	ScoreSSHBruteforceBolt	Heuristics based scoring for potential SSH brute force attempts.
score-http-brute	ScoreHTTPBruteforceBolt	Heuristics based scoring for potential HTTP or HTTPS brute force attempts.
score-tcp-hosted-service	ScoreServiceBolt	Negative scoring for traffic to unknown services on local IP addresses.
score-tcp-remote-service	ScoreServiceBolt	Traffic to known remote services (e.g. DNS or SMTP).
score-bad-udp-ports	ScoreInsecurePortConversationBolt	Negative scoring for flows on known insecure UDP ports.
score-bad-tcp-ports	ScoreInsecurePortConversationBolt	Negative scoring for flows on known insecure TCP ports.
score-bad-udp-traffic	ScoreUnknownPortConversationBolt	Negative scoring for flows on unknown (i.e. non standard) UDP ports.
score-bad-tcp-traffic	ScoreUnknownPortConversationBolt	Negative scoring for flows on unknown (i.e. non standard) TCP ports.

On the surface this may appear to be the case but there are differences. In this case, the first bolt is looking for traffic that explicitly does not match what is hosted on the servers in question, while in the second case all traffic on ports known to be insecure is scored.

4.10 Scored Flows Analysis

The topology ingests flows and as part of the processing attempts to match unidirectional flows together forming network conversations. For this reason, the output data is referred to as conversations instead of flows. There are three output streams from our topology: the scored conversation, the scored conversation tags and the scan candidate records. The first two sets of data were analysed further in order to evaluate the effectiveness of the solution. Two approaches were used for the evaluation and these are both briefly described in this section.

4.10.1 Visualisation

For purposes of visualisation flows persisted by the PersistScoredBoltToKafka bolt were ingested into a Elasticsearch/Logstash/Kibana stack. This stack enables rapid ingestion and processing of large volumes of log data for purposes of visualisation or notification purposes. A number of dashboards were configured in order to view different aspects of the scored flow data. The dashboards allowed for a dynamic interrogation of the data enabling drilling down or filtering on aspects such as time periods, geolocation, ports, protocols and hosts.

4.10.2 Static Analysis

The source flow data, the scored conversations and the tags were loaded into a Postgres database for further analysis. Using standard SQL the data was filtered, summarised and then inspected or charted for insights into the results. This process in tandem with the dynamic visualisation described allowed for quick identification of points of interest for further investigation.

4.11 Summary

In this chapter, the implementation of the scoring topology has been described in detail. The contents cover the journey that the flow data takes, from collection through to processing and on to analysis. The bolts described in this chapter have been chosen to illustrate a wide cross section of what is possible. Production implementations may use less bolts or implement new ones as required. The final topology presented was the result of numerous experimental runs, but as with the bolts it can be easily changed to suit differing environmental requirements. In the following chapter the results of running the implementation with a large data set is presented. Both the scoring outcomes as well as the framework performance is discussed in detail.

Chapter 5

Results and Discussion

In this chapter, we discuss the data sample, the processing methodology, the characteristics of our data before and after processing, and finally, we discuss the outcome of the flow scoring and tagging process. In section 5.1, the sample data and its processing is discussed along with some detail on the performance characteristics of the Storm topology. Section 5.2 describes the processing run presenting statistics on the performance of the Storm topology and the operating system. In sections 5.3 the scored data is examined and points of interest uncovered by the scoring process are noted. These are then further investigated and documented in section 5.4.

5.1 Data Processing

This section outlines the data that was used and the configuration of the Storm topology for processing it. Data is presented from tests that were undertaken in order to configure the topology for optimal throughput. Operating system statistics from the final processing run are also discussed.

5.1.1 Sample Data

The NetFlow data used for testing Themis was collected on the Internet gateway of a /24 network. The number of active hosts on the network varies from time to time, but is generally around 60. These hosts function primarily as mail servers, proxy servers,

firewalls and Internet gateways, in addition to acting as NAT gateways for a number of school networks with around 2500 users in total. The data set consists of 1,475,333,299 NetFlow records collected between 9:35 on 2017-01-10 and midnight on 2017-03-31 (an average of 211.8 flows per second). The records are unidirectional, recording only traffic seen in one direction with each network conversation resulting in two records in the data set. All IP addresses from the internal network were anonymised as described in section 4.3 by mapping hosts in the inside /24 to corresponding hosts in the 192.0.2/24 netblock (as per RFC5737).

5.1.2 Timestamp Accuracy

During previous work with nfcapd data files issues with flow timestamps were encountered (Sweeney and Irwin, 2017). The cause of problem was found in a bug report relating to how time is handled by nfdump (Swarankar, 2010). Specifically, there is a overflow issue that will lead to an offset of approximately 50 days (4,294,967,296 msec) in some of the date attributes. The symptoms of this bug appeared in the sample data on a small percentage of the data causing some of the flow start times or end times to fall significantly outside the collection period. The current implementation of Themis does not take start and end times into consideration, and only requires a single timestamp for each record. Where a records data was correct, the flow start time was used as the timestamp. For buggy flow data, the timestamp was set to the start time only if it fell within the collection period. If this value was invalid, then the end time was chosen. No records were found with both values being incorrect.

5.1.3 Topology Scoring Configuration

In this section, the Storm bolt configuration used for the bolts during the full processing run are described and discussed. It is important to understand the aim of each bolt instance, the bolt configuration and the intended outcome in the scoring and tagging process. The details of flow scoring are covered in 2.1.4 and the list of bolts in the topology can be found in table 4.1. This sections describes the bolt setup in detail.

As discussed in Chapter 4, there are three categories of bolts: enrichment bolts, utility bolts and scoring bolts. Scoring bolts are themselves divided into a further two categories:

those that test for badness and those that test for goodness. As part of the configuration of Themis, the scoring bolts required an individual score to be assigned for each type of test. It was decided to use a value in the range 0 and 100, representing a percentage of confidence in the scoring test. The higher the value, the more confidence in the test outcome and vice versa. Tests such as the white list one using the IP addresses of well-known services such as Google, YouTube and Facebook were assigned a high value indicating a strong level of confidence that the source data was accurate (the IP addresses) and the associated traffic was most likely benign (i.e. good). There is a strong probability of the latter holding true as we are merging the unidirectional flows into network conversations making the likelihood of spoofing from these addresses near zero. For scoring traffic flows to dark (or unassigned) IP address space, a lower score was assigned as the likelihood of the traffic always being intentionally malicious is considered minimal. The tags assigned during scoring are also described below and are made of a category and a code separated by a “-”. A summary of this configuration is presented in table F.1 in appendix F.

The first class of bolts that we will consider are those that do scoring based off static lists (see section 4.6.4). These lists are sourced from third party websites and are usually used as either whitelists or black lists. The first such list is the IP address ranges of known services such as Google, YouTube, etc, used by the open NDPI project for traffic classification. A whitelist was constructed from these addresses and a *ScoreGenericIPListBolt* was configured using this list. The flows with IP addresses in the list were assigned a goodness of 100, along with the a tag of IP_LIST-NDPI_GOOD. The high goodness score was used to indicate a strong level of confidence that the traffic to these servers was highly likely to be benign.

Two static blacklists were also used; one using data from the AlienVault site and one from the Emerging Threats site. In each case, a *ScoreGenericIPListBolt* was configured with the list data and both were configured to score matching flows with a badness of 50. This median value was chosen to represent the fact that the information in the freely available lists may be out of date as the lists were downloaded in August, while the actual data in our sample was over the period January to March. The time mismatch was deemed acceptable for our proof of concept. The tags for the flows that matched in these two bolts were IP_LIST-ALIENVAULT and IP_LIST-EMERGING_THREATS respectively.

The next class of bolts configured are the ones that score network conversations (i.e. where a bidirectional flow of traffic has taken place - see section 4.6.6). Scoring looks

Table 5.1: Port scoring tags

Port Specific Scoring				
Protocol	Type	Tag	Score	Description
UDP	Insecure	PORT_LIST-INSECURE_UDP_TRAFFIC	50	Traffic observed to insecure UDP services
TCP	Insecure	PORT_LIST-INSECURE_TCP_TRAFFIC	50	Traffic observed to insecure TCP services
UDP	Unknown	PORT_LIST-UNKNOWN_UDP_TRAFFIC	70	Traffic observed between unknown UDP ports
TCP	Unknown	PORT_LIST-UNKNOWN_TCP_TRAFFIC	70	Traffic observed between unknown TCP ports

Table 5.2: Service scoring tags

Service Specific Scoring				
Protocol	Tag	Good Score	Bad Score	Description
UDP	HOSTED.SERVICES-UDP	100	50	Known hosted UDP services on the internal network
TCP	HOSTED.SERVICES-TCP	100	50	Known hosted TCP services on the internal network
UDP	REMOTE.SERVICES-UDP	50	0	Expected traffic to remote UDP services
TCP	REMOTE.SERVICES-TCP	50	0	Expected traffic to remote TCP services

at the ports in question, a minimum number of packets and the protocol. Instances of these bolts can be used to flag potentially insecure traffic (e.g. FTP, TCP or RLOGIN) or unknown network flows (e.g. conversations between high numbered, unassigned ports). Four instances of this class of bolt were configured as shown in table 5.1. Insecure traffic was deemed less suspect than unknown traffic as some organisations still make use of these protocols. For all four cases a minimum of three packets was required in the conversation before the scoring was triggered.

Four instances of *ScoreServiceBolt*'s (see section 4.6.7) were configured to score remote and local traffic that corresponded to known or expected services (e.g. HTTP, DNS, SMTP, IMAP, etc). The primary purpose of these bolts is to score expected traffic with a good score and thereby allow for the filtering out of benign traffic. Information on the roles of the various hosts on the inside network were gathered and used to build a expected traffic profile for a number of the hosts on the inside network. This profile was then used to configure these bolts as per table 5.2. The first two bolts were configured to score expected conversations from services such as SMTP servers, DNS servers, HTTP servers, etc. hosted on the internal servers. Matching conversations were scored with a goodness of 100, while conversations that did not match were scored with a badness of 50 (negative scoring). The lower badness score was chosen to take into account that there may be valid services on the internal hosts that were not known at the time of configuration. The second two bolts were configured to score expected traffic conversations to remote services. This allowed, for example, traffic from proxy servers to remote web servers to be scored as good. These bolts applied a goodness value of 100 to all matching traffic indicating a high level of confidence that this was expected network conversations.

The custom bolts implementing logic to detect SSH and HTTP/HTTPS brute force password guessing (see 5.4.6) were each assigned a badness score of 90, as their heuristics are based on prior work and have been tested. The tags assigned were BRUTE_FORCE-SSH and BRUTE_FORCE-HTTP. The dark IP bolt (section 4.6.3) was assigned a score of 40 representing a low confidence that traffic to unassigned IP addresses is suspicious. The tag for this bolt was set to DARK_IP. The country based scoring (see section 4.6.2) was configured to score any traffic from China or Russia with a badness of 80 and the tag SUSPECT. These countries, and their high score, were chosen as they are often considered the two top state actors where cyber threats are concerned (Smeester and Associates). The *ScoreIntelMQBolt* (section 4.6.9) was assigned a value badness score of 70 and the INTELMQ tag. The reasonably confident value was chosen as the source data came from a live and well-curated list of threat intelligence. Finally, the *ScorePossibleScanBolt* (section 4.6.8) was assigned a badness of 60 and the tag POSSIBLE_SCAN-SYN_ONLY.

A bolt was added to the beginning of the topology to discard flows that were considered “uninteresting”. Three categories of flows matched this definition:

- **Uncommon Protocols:** For purposes of this work only ICMP, UDP and TCP traffic was examined. All other traffic was discarded.
- **Mirai Traffic:** The inside network was configured with a honeypot for potential Mirai botnet traffic (Dobbins, 2016; Herzberg *et al.*, 2016). This created false traffic flows that could affect the analysis. In order to filter out false positives all traffic to TCP port 23 or 2323 was discarded.
- **Internal Traffic:** As the focus is on external threats, any traffic conversations exclusively between hosts on the inside network was discarded.

Due to the POC nature of this project a full justification for the particular values chosen for scoring is a difficult process and considered out of scope. There is much scope for an investigation into a more reasoned scoring model. In addition, it would be valuable to investigate a feedback mechanism that could evaluate the effectiveness of a chosen model and allow for adjustments over time.

5.1.4 Processing Environment

All processing took place on a server with the configuration as shown in table 5.3.

Table 5.3: Processing Environment

Processing environment	
Attribute	Value
OS	Ubuntu 16.04.2 LTS
CPU's	4 x Intel(R) Xeon(R) CPU E3-1230 v6 @ 3.50GHz
RAM	16GB
Application	Version
Apache Zookeeper	3.4.9
Apache Kafka	0.10.2.0
Apache Storm	1.0.3
IntelMQ	1.1.0
Logstash	5.0.0-beta1
ElasticSearch	5.5.1
Kibana	5.5.1
PostgreSQL	9.5
Python	2.7
Redis	3.0.6

5.1.5 Apache Storm Configuration

Storm's scaling and concurrency model is discussed in section 4.1.2 (see also figure 4.6). Generally speaking, there are three parameters that influence the parallelism:

- The number of workers. This value is set for the topology and is the number of JVM instances to be launched to run the topology. The convention for this is to use one worker per node. In our case, we have a single node and therefore only configure a single worker.
- The number of executors. This value is set at a per bolt level and is the number of threads per worker to launch in order to execute the bolt.
- The number of tasks. This value is set at a per bolt level and is the number of bolt instances to run in the topology. The tasks for a bolt will all be run in the bolts executor threads. This means that the number of executors is always \leq number of tasks. The default is to run one task per executor.

Another setting that affects the performance of the topology is the number of outstanding tuples allowed at any point in time. In order to ensure reliability, the tuples consumed and generated by bolts are tracked by the Storm framework (this is called the tuple tree)

with each bolt along the way acking the tuples as they are processed. If the entire tree of messages generated from a tuple emitted by a spout is not processed within a timeout, then that originating tuple is considered failed for processing purposes. The number of outstanding (or unacknowledged) tuples is a parameter set on the spouts that is used to control the throughput load on the topology. If this is set too high for the processing capabilities of the topology, then it may be overwhelmed and tuples processing will time out. If it is set too low, the topology may not be fully utilised.

5.1.6 Storm Performance

In order to evaluate the performance of the topology the following statistics were collected and compared:

- The number of flows ingested per second. This was measured using a *PersistTimerBolt* located at the beginning of the topology to count tuples emitted from the spout.
- The number of flows scored per second. This was measured using a *PersistTimerBolt* located at the end of the topology to measure the final count of scored flows.
- The per-bolt capacity measure. This statistic is tracked by Storm on a per bolt level and is an indication of how busy a bolt is. A value close to or greater than one indicates that the bolt is running as fast as possible and is overloaded. By tracking this value during a processing run, it is possible to determine which bolts are doing the most work and potentially could utilise more resources.
- The latency of a tuple. This value is taken from the Storm UI and represents the time taken on average for a tuple to be completely processed by the topology.

The first two values give an indication of the topology throughput (how many flows are processed per second), while the last one is the topology latency (how fast flows are scored). The data collected is presented later in this chapter.

During the development and testing phase, the topology was run with a number of different permutations of the settings in order to try and determine the ideal set for the final processing run. The test data consisted of five million tuples and the statistics were collected and reviewed. One of the first tests consisted of running the topology with one

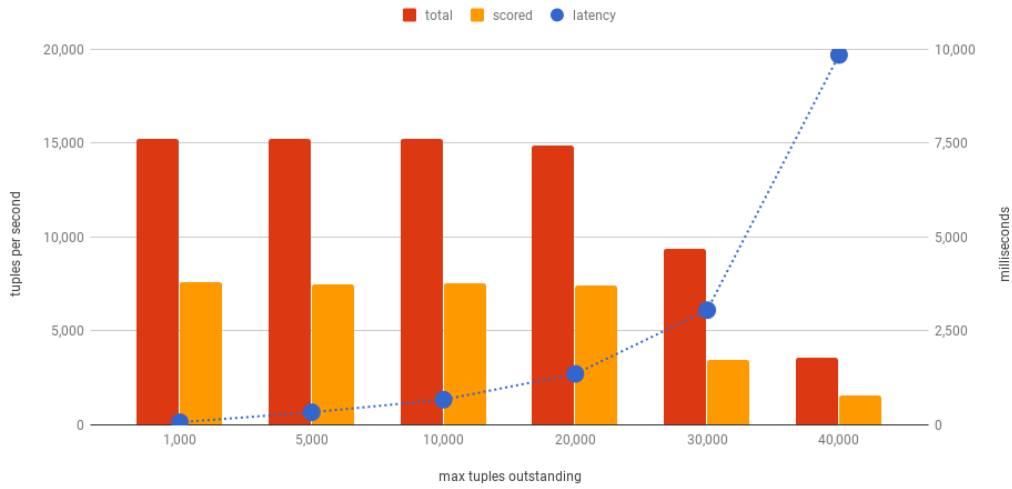


Figure 5.1: One executor, one task per bolt

executor and one task per bolt using an outstanding tuples value ranging from 1,000 to 40,000. The results of this are shown in figure 5.1. The bars show the number of flow records processed per second (total and scored), while the circles show the per flow latency (average processing time) in milliseconds. From chart 5.1 it can be clearly seen that changing the outstanding tuples value has little effect on throughput until a value of 30,000 and after that performance degrades significantly. An interesting point to note is that the latency increases as the outstanding tuples value is increased. This is likely due to more tuples being queued for processing and adding overhead to the topology.

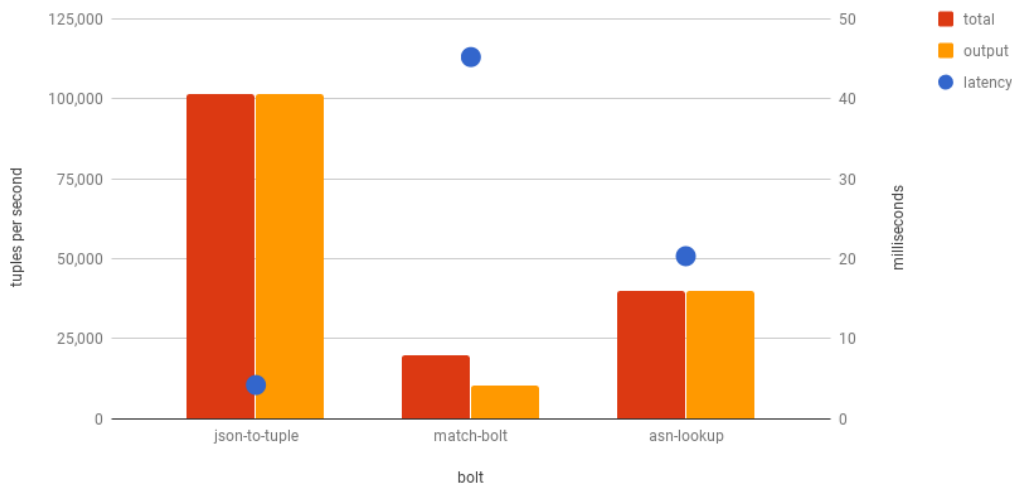


Figure 5.2: Bolt comparisons (1,000 tuples outstanding)

During the testing, the bolt capacity measures were monitored and it was noted that the

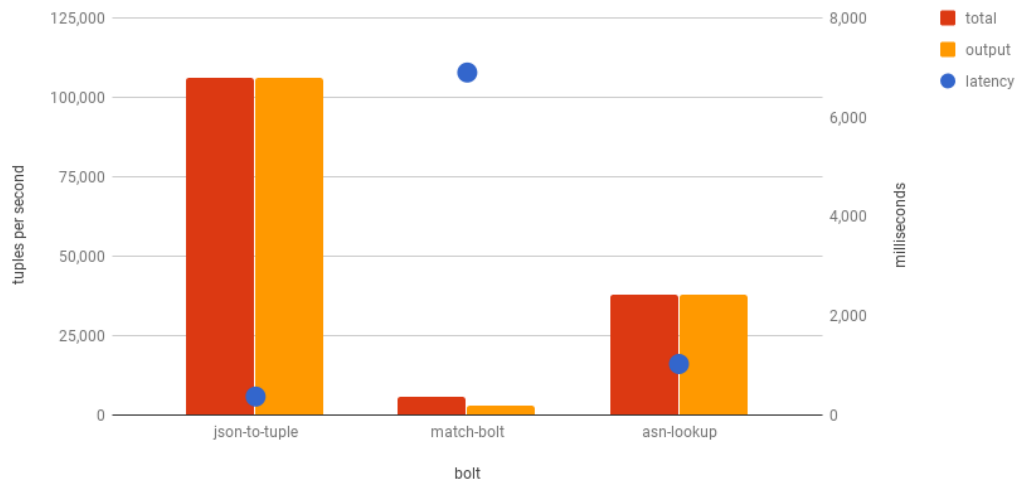


Figure 5.3: Bolt comparisons (40,000 tuples outstanding)

MatchFlowBolt had a values of one and greater across a number of runs. Three stripped down topologies were deployed all containing the Kafka spout and the *JSONtoTupleBolt* (see section 4.5.1). The first one had no other bolts configured, the second and third were deployed with the *ASNBolt* (section 4.5.2) and the *MatchFlowBolt* (section 4.8.3) respectively. The *ASNBolt* was chosen as it had the second highest capacity score. The graph in figure 5.2 compares latency and throughput for each of these topologies using a maximum outstanding value of 1,000 tuples. In figure 5.3, the same tests were run, but with a maximum outstanding value of 40,000 tuples. Comparing the charts, it can be seen that the *MatchFlowBolt* appears to be a bottleneck, consistently showing the lowest throughput with the highest latency. This bolt is responsible for matching flows and makes use of a number of caches for tracking flows and the overhead of this tracking and matching is most likely the cause for the lower performance of this bolt.

To compensate for this the scoring topology was modified to allow more *MatchFlowBolt* thread instances to be configured than other bolts. After some more testing it was found the following configuration gave a minor gain in throughput with minimal effect on latency:

- Eight instances of the *MatchFlowBolt*, each running in its own thread (8 executors and 8 workers).
- For all other bolts, four instances, each running in its own thread (4 executors and 4 workers).

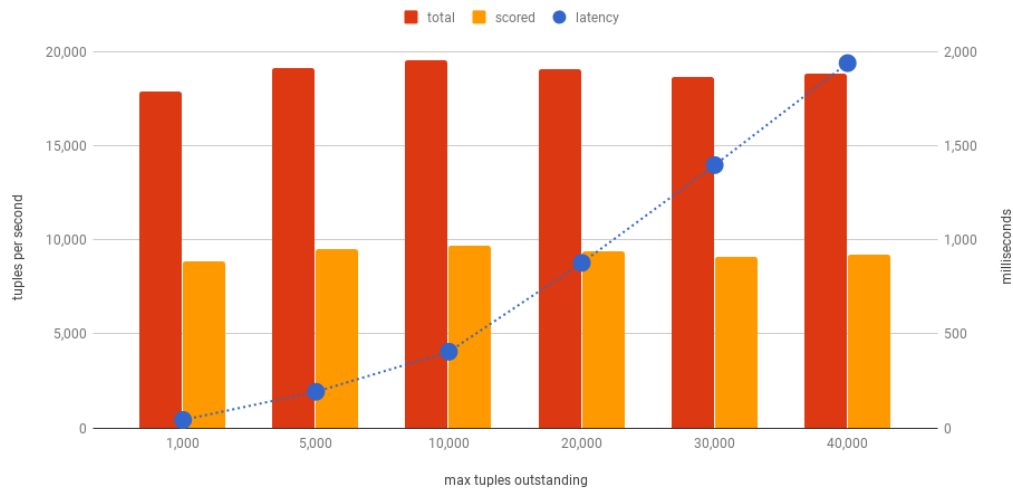


Figure 5.4: Four executors, four tasks per bolt. Eight for MatchFlowBolt

The run time statistics are shown in figure 5.4. At this point it was decided to use this configuration for the final processing run.

5.2 Processing Run

The sample data was queued in Kafka and a topology deployed with the bolt and topology configuration described in this sub section. The topology was allowed to run until all the flows were processed and the resulting scored conversations, conversation tags and potential scan candidate flows were queued again in Kafka. A total of 1,475,333,299 flow records were ingested. From this, 750,089,603 scored conversations were produced along with 1,137,317,767 tags. Finally, 98,243,303 scan candidate flows were identified and emitted into a separate Kafka topic.

The total processing time for the 1,475,333,299 flow records was 22 hours and 19 minutes. During the processing run samples were taken of operating system statistics, Storm statistics and the topology throughput. The statistics presented here are from an eight-hour sample period during the processing run.

In figure 5.5 the CPU times from the server during processing are presented. Three key values are shown: user time (time spent running non-kernel code), system time (time spent running kernel code) and idle time. The key value is user time as this represents how

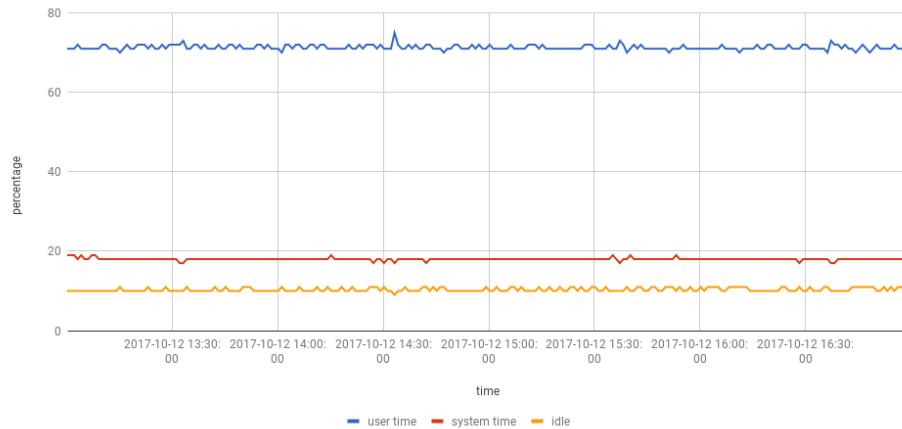


Figure 5.5: OS CPU Counters

much time is spent running applications on the sever. The majority of this time would be taken up by the services required to run and support the running of the topology. From the chart we can see that this value remains steady at around 71% of total CPU time (average 71.3 with a standard deviation of 0.6). Idle time is an indication of spare CPU cycles and from the the chart it is clear that the processing is not consuming all available resources. This is an indication that we could possibly increase the parallelism configuration of the topology in order to increase throughput. Note that the busy cycles are not just being spent in the Storm topology - all the supporting applications such as Redis, Kafka, etc. are running on the same server and are also consuming resources.

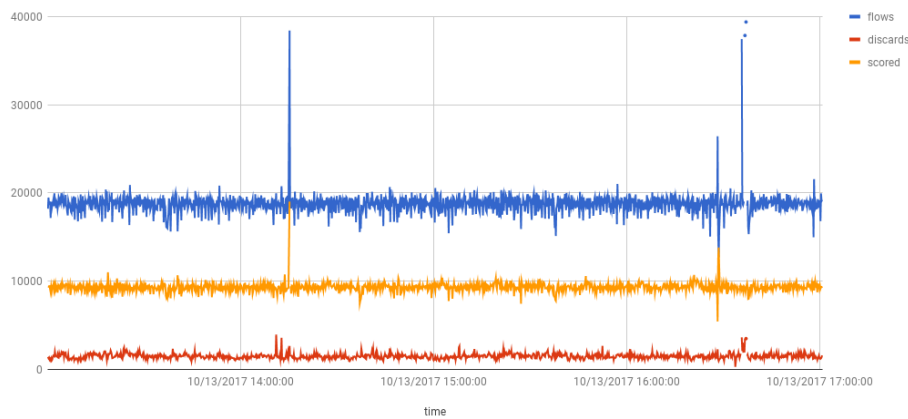


Figure 5.6: Processing throughput

In figures 5.6 and 5.7 we can see the processing throughput and latency statistics. In the throughput chart we see throughput counters for flows ingested, flows discarded (see

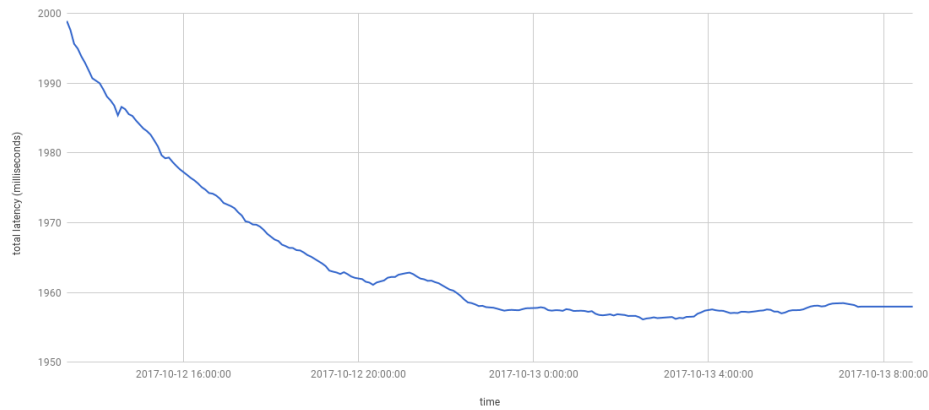


Figure 5.7: Processing latency

Table 5.4: Throughput statistics

Throughput stats				
Attribute	Min	Average	Max	StdDev
Total flows	12,705	18,756	39,399	1,336
Discarded flows	255	1,463	3,912	281
Scored conversations	5,414	9,291	19,015	482

the paragraph on discards in 5.1.3) and scored conversations emitted. Flow processing throughput remains steady throughout at around 18,800 flows per second with a discard rate of 280 flows per second and a scoring rate of 9,290 flows per second. More details are presented in table 5.4. The latency for the processing shows a very slight improvement over time until a plateau of 1,961 milliseconds is reached. The improvement over time can possibly be attributed to JVM HotSpot compilation, as there were no other changes or events on the host during the processing run.

Finally, figure 5.8 gives a comparative breakdown of some performance metrics for each Bolt. The bar chart shows the average processing latency (time spent per tuple) in milliseconds, while the line shows the bolt capacity. As discussed previously, capacity is a measure of how busy a bolt is with a value of one representing 100% busy. The MatchFlowBolt stands out as a clear bottleneck in the topology. Also interesting to note is that any bolts that interact with external data also incur penalties. This would suggest that investigating more in memory caching options may improve performance.

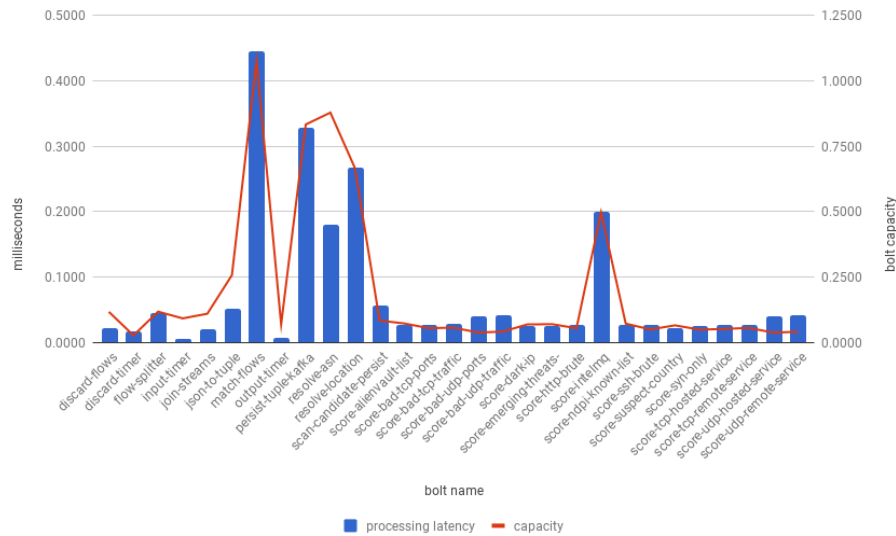


Figure 5.8: Bolt performance

5.3 Scoring and Tagging Results

In this section, the results of the flow processing are presented and discussed. Due to time and resource constraints, only a subset of the scored conversation data was analysed in any great depth with a view to demonstrating the correct functioning of the system. In this section, a look is taken at how the data has changed in nature in terms of size and quality. Then a subset of data is examined from a number of different perspectives, focusing on different attributes of the scored data in order to identify potential threats. In the next section, these potential threats will be examined in more detail in order to determine whether our processing has achieved its goal of assisting in threat detection. Note that only subset of possible incidents are picked out for illustration purposes, as a full analysis is beyond the scope of this work. The output of the scoring and tagging process are termed conversations going forward. The reasoning for this is that the unidirectional flows have been matched where possible and are now representative of a network conversation as opposed to a flow of traffic.

5.3.1 The Data Funnel

The flow processing resulted in an output of 750,089,603 scored conversations. These results had a number of permutations of scores as shown in table 5.5. In order to do

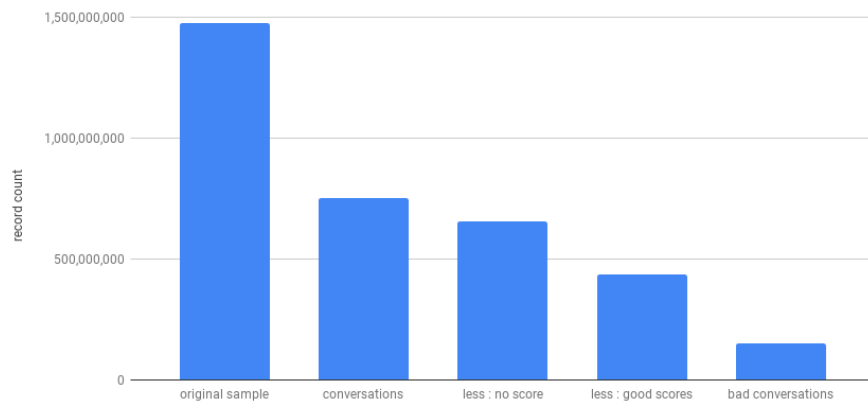


Figure 5.9: Data reduction

Scoring permutations			
Category	Quantity	Percentage	Description
No Score	96,643,312	12.89%	Records with neither a good score or a bad score.
Only Goodness	219,469,833	29.27%	Records that have only a good score.
Only Badness	151,158,370	20.16%	Records that have only a good score.
Mixed Score	282,477,095	37.68%	Records that both good and bad score.

Table 5.5: Scoring categories

further analysis of the data within the project resource limits, it was decided to focus on the potentially worst cases - the records with only bad scores which account for 20% of the output data (hereafter referred to as the “bad conversations” or “bad traffic”). It is interesting to note how the quantity of data initially presented has been filtered down to a more manageable amount through the scoring process. In figure 5.9, we can see the reduction in size of the data illustrated as a succession of bars from left to right. The final data set size to be reviewed is only 10% of the size (in record count) of the original sample of NetFlow data.

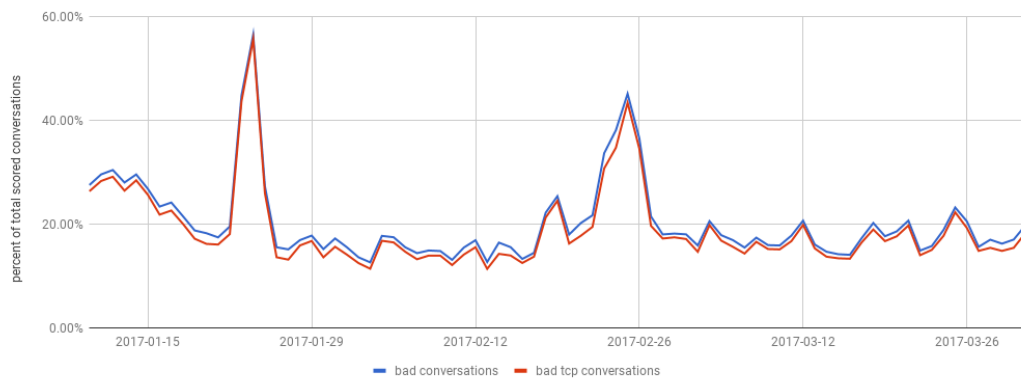


Figure 5.10: Bad conversations as percentage of total

```
1 {  
2   "src_ip_int": 1757543728,  
3   "protocol": 6,  
4   "bytes": 240,  
5   "src_port": 39968,  
6   "packets": 4,  
7   "src_ip": "104.193.253.48",  
8   "flags": "...S.",  
9   "dst_port": 45554,  
10  "dst_ip_int": 3289772733,  
11  "dst_ip": "192.0.2.189",  
12  "start_timestamp_unix_secs": 1486834707,  
13  "end_timestamp_unix_secs": 1486834707,  
14  "id": 554505213  
15 }
```

Listing 5.1: Flow record input

Plotting the number of bad conversations per day as a percentage of the total, we can see that the value of 20% is more or less constant except for two notable spikes on the 24th January and the 23rd of February (figure 5.10). By plotting the percentage of TCP only bad conversations on the same chart we can see that this profile closely matches that of the bad conversations. The spikes in question are clearly TCP related. By simply considering the amount of bad conversations we can already identify events in time that require investigation to determine threat potential.

5.3.2 Data Enrichment

A key feature of Themis is the enrichment of the source data allowing for detail analysis, alerting and reporting. In listing 5.1, an example of an input JSON record is shown. Listing 5.2 shows the data now available for this same record (also in JSON format).

As can be seen the richness of the information has been significantly enhanced with added information such as geolocation, AS numbers, matched flow records (forming a conversation), as well as augmented scoring and tag data. In effect, the amount of data has been reduced, but the richness has been significantly increased. Note that the geolocation data is only given for the external or remote hosts in the scored records.

5.3.3 Traffic Statistics

The first statistics reviewed are the general stats of the scored sample in terms of the volume of traffic and number of conversations that are bad. In all following discussions the

```

1 {
2   "badness": 200,
3   "packets_recv": 4,
4   "dst_as": 14576,
5   "start_timestamp_unix_secs": 1486834707,
6   "flags": "...S.",
7   "type": "scored_flow",
8   "dst_ip": "104.193.253.48",
9   "src_ip": "192.0.2.189",
10  "protocol": 6,
11  "packets_sent": 0,
12  "flow_id": 554505213,
13  "end_timestamp_unix_secs": 1486834707,
14  "goodness": 0,
15  "src_as": 2018,
16  "timestamp": 1486834707,
17  "geoip": {
18    "timezone": "America/Los_Angeles",
19    "ip": "104.193.253.48",
20    "latitude": 37.5497,
21    "continent_code": "NA",
22    "city_name": "Fremont",
23    "country_name": "United States",
24    "country_code2": "US",
25    "dma_code": 807,
26    "country_code3": "US",
27    "region_name": "California",
28    "location": {
29      "lon": -121.9621,
30      "lat": 37.5497
31    },
32    "postal_code": "94539",
33    "region_code": "CA",
34    "longitude": -121.9621
35  },
36  "event_time_str": "2017-02-11T17:38:27.0Z",
37  "bytes_sent": 0,
38  "src_port": 45554,
39  "@timestamp": "2017-10-20T09:59:11.308Z",
40  "flow_direction": "INBOUND",
41  "dst_port": 39968,
42  "bytes_recv": 240,
43  "matched": false
44 }
45 {
46   flow_id    : 554505213,
47   score_name : POSSIBLE_SCAN ,
48   score_category : SYN_ONLY ,
49   score      : 60,
50   "is_bad_score" : t
51 }
52 {
53   flow_id    : 554505213,
54   score_name : PORT_LIST ,
55   score_category : UNKNOWN_TCP_TRAFFIC ,
56   score      : 70,
57   "is_bad_score" : t
58 }
59 {
60   flow_id    : 554505213,
61   score_name : INTELmq ,
62   score_category : malware : malicious code ,
63   score      : 70,
64   "is_bad_score" : t
65 }

```

Listing 5.2: Enriched conversation record

traffic is presented from the perspective of the internal network. Sent refers to outbound traffic (i.e. sent from the network under study) and received refers to inbound traffic (i.e. received from the outside). For the purposes of this exercise, we only look at UDP and TCP traffic when breaking out the protocols from the total traffic.

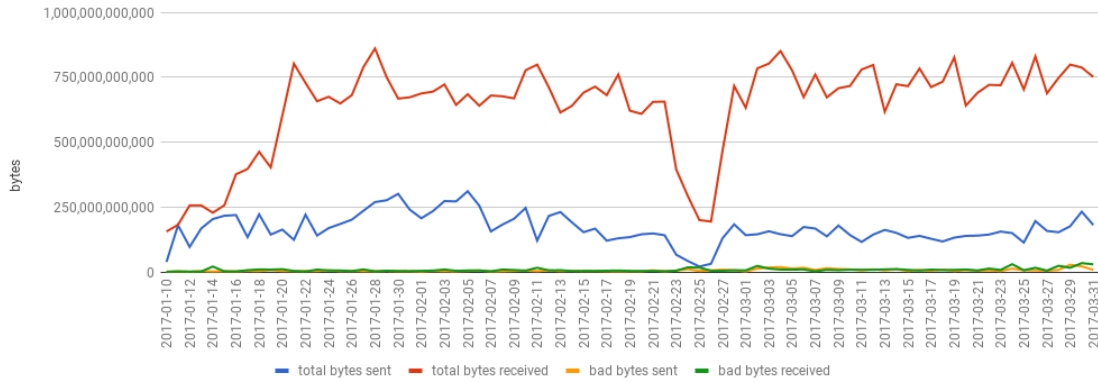


Figure 5.11: Traffic utilisation comparison

In figure 5.11, the bytes sent and received of the total scored sample and the bad traffic subset are plotted for comparison. As can be seen, the bad sample makes up a small part of the overall traffic and the chart provides little information of value. Despite the fact that the bad conversations make up 20% of the total conversations, they only account for 3.8% and 1.3% of the total bytes sent and received respectively. In order to better understand the impact of the bad conversations, we need to look at the data in terms of ratios - how much bad traffic there is compared to the overall amount.

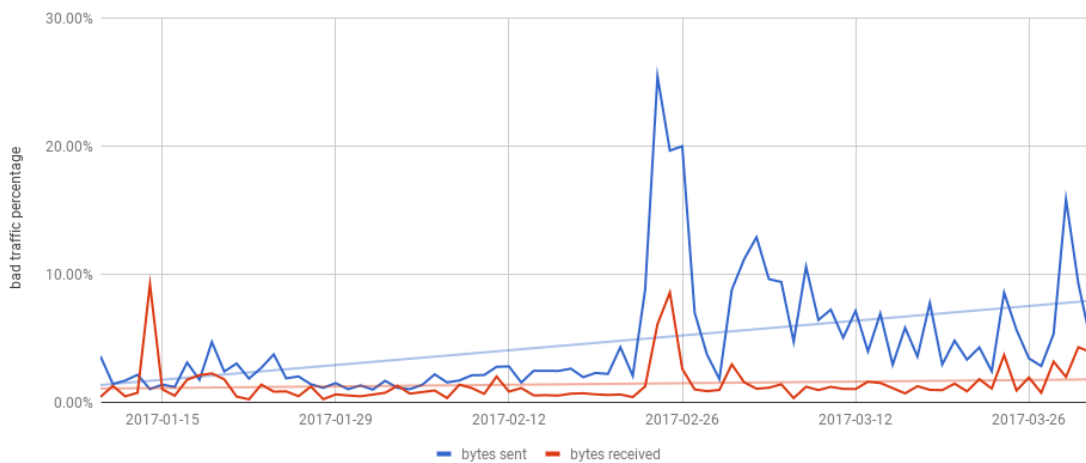


Figure 5.12: Bad sample as percent of total

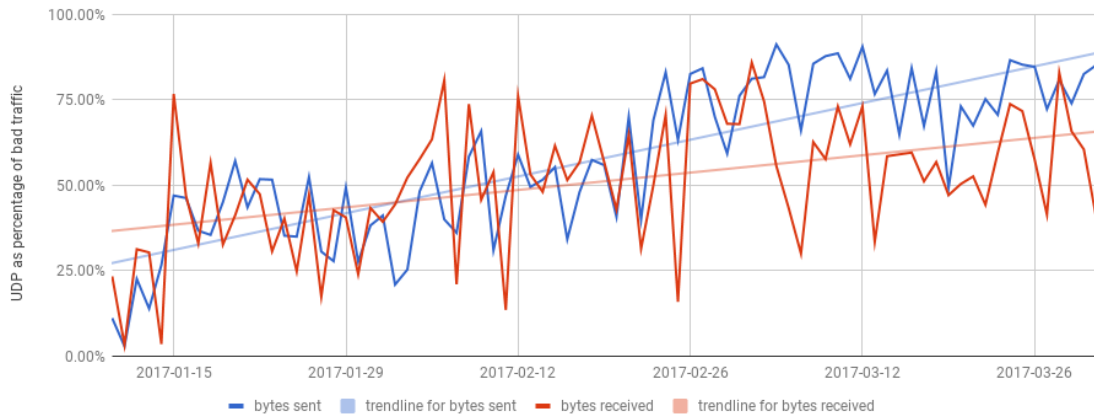


Figure 5.13: Bad UDP sent traffic as percent of total

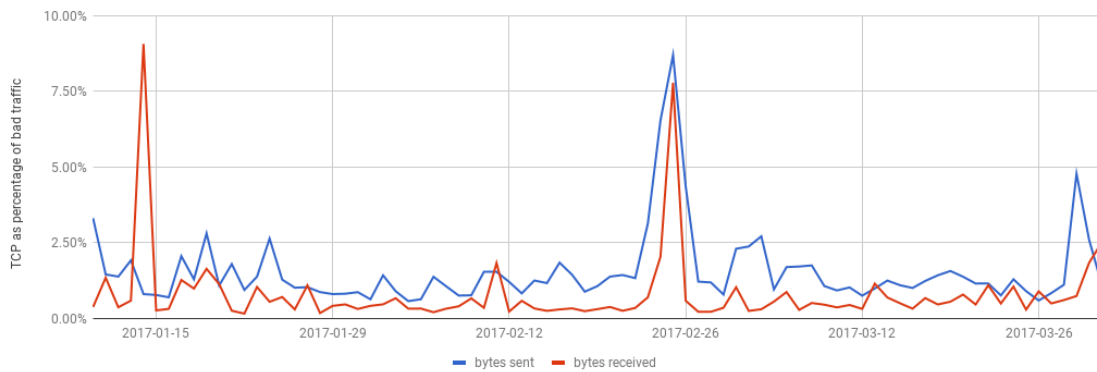


Figure 5.14: Bad TCP sent traffic as percent of total

In figure 5.12, the bad sample values are plotted as a percentage of total traffic and trendlines are added. Here we can immediately see two areas of interest: a definite increase in the volume of bad traffic in the second half of the collection period and a spike of bad incoming and outgoing traffic volumes around the 25th of February. Breaking out the constituent protocols, we can see that UDP traffic is largely responsible for the increase in bad traffic from midway through the sample period (figure 5.13). If we take a look at the TCP only chart (figure 5.14), we can see that the spike on the 25th of February was primarily caused by TCP traffic tagged as “bad”. This event has a possible correlation with the bad conversation spike observed in figure 5.10 on the same date. Finally, a spike in incoming bad traffic can be seen on the 14th of January.

There are two other sets of data that can be considered from a simple network traffic point of view: top hosts by traffic volume and top ports by traffic volume. These two views can further be broken down into top senders and top receivers respectively. We can

Table 5.6: Top bad traffic sources and destinations by inside host

Top Hosts Inside - TCP					
Host	Bad %	Bytes Sent From	Host	Bad %	Bytes Received On
192.0.2.144	99.88%	4,940,539,282	192.0.2.66	99.99%	200,492,970
192.0.2.137	99.68%	5,115,636,219	192.0.2.131	100.00	316,603,381
192.0.2.180	96.46%	10,440,746,762	192.0.2.181	100.00%	660,958,209
192.0.2.190	88.73%	31,173,459,387	192.0.2.137	95.77%	1,291,354,286
192.0.2.189	91.12%	32,473,062,551	192.0.2.144	99.98%	2,529,288,651
Top Hosts Inside - UDP					
Host	Bad %	Bytes Sent From	Host	Bad %	Bytes Received On
192.0.2.9	100.00%	226,771	192.0.2.186	99.93%	14,447,903
192.0.2.66	100.00%	8,072,886	192.0.2.66	99.91%	19,720,984
192.0.2.241	99.91%	9,526,099	192.0.2.241	99.96%	20,492,780
192.0.2.242	100.00%	31,688,939	192.0.2.242	99.99%	58,774,798
192.0.2.130	98.63%	124,524,793,906	192.0.2.130	96.08%	47,608,620,450

look at the traffic across these combinations from the point of the inside of the network (i.e. the network of interest) or from the outside of the network (i.e. the Internet). The concept of inside vs. outside is explained in detail in section 4.2.1. Simply looking at total bytes sent or received in each case will give little insight. In order to better identify potential issues leveraging our scoring, we can filter this data by calculating how much of the traffic volume is from our bad conversations and reflecting it as a percentage of the total traffic in its class. Combinations with a higher percentage of bad traffic are more significant and can be investigated further.

In table 5.6, the internal hosts with the top bad traffic volumes (calculated as a percentage of total traffic) are listed. The values in the Bad column indicate what percentage of the total traffic volume came from conversations in our bad sample. A value of 100% indicates that all traffic for the host in the direction given was scored as bad. The Bytes column is the traffic volume from our bad conversations. If the percentage column is 100%, then this value represents the entire flow of data for the host in the direction given. Lower percentage values indicate that the traffic volume listed was not all the traffic seen for this host. The traffic direction is classed as volume sent to the Internet and volume received from the Internet respectively and this is further broken down into TCP and UDP traffic. The most striking entry in the table is the UDP traffic statistics for host 192.0.2.130. The volume of outbound traffic exceeds the next closest entry by four, accounting for 24% of all bad traffic sent from the inside network. Further investigation found that the majority of this traffic was Skype-related traffic to the host 104-44-200-136.relay.skype.com which was tagged as an unknown UDP conversation. A second item to note is that for all of the hosts in the table, over 90% of their traffic is deemed bad. This may not necessarily be a problem, as in this table we are flagging all hosts with bad scores. The scores themselves

Table 5.7: Top bad traffic sources and destinations by outside host

Top Hosts Outside - TCP					
Host	Bad %	Bytes Sent To	Host	Bad %	Bytes Received From
93.190.141.3	99.94%	2,615,709,645	185.163.109.22	100.00%	4,101,666,925
137.74.0.124	99.81%	3,119,575,961	197.221.20.242	96.08	4,172,488,438
129.232.191.147	100.00%	5,058,840,827	185.21.216.148	100.00	5,647,743,252
197.221.20.242	98.95%	5,300,249,587	163.172.85.64	100.00	5,665,946,586
208.91.113.203	99.99	16,374,624,892	5.79.105.26	99.86	20,627,654,678
Top Hosts Outside - UDP					
Host	Bad %	Bytes Sent To	Host	Bad %	Bytes Received From
104.44.200.47	100.00%	6,081,071,747	169.239.44.14	100.00%	8,399,613,740
157.240.1.51	100.00%	8,318,044,544	188.26.196.255	100.00%	8,660,520,350
104.44.200.142	100.00%	16,758,066,678	172.111.197.130	100.00%	10,880,290,252
169.239.44.14	100.00%	27,098,936,615	45.221.64.2	100.00%	21,851,874,708
104.44.200.136	100.00%	76,145,175,168	104.44.200.136	100.00%	43,000,777,408

Table 5.8: Top bad traffic sources and destinations by inside port

Top Ports Inside - TCP					
Port	Bad %	Bytes Sent From	Port	Bad %	Bytes Received On
3389	99.76%	1,215,849,652	31743	99.15%	945,248,866
21299	99.85%	1,277,776,367	29900	98.39%	1,240,210,308
35790	98.33%	1,497,548,155	21465	98.06%	1,260,568,154
22	99.66%	3,438,780,880	22	99.72%	2,530,159,077
45554	98.34%	62,198,160,867	45554	90.41%	15,092,991,874
Top Ports Inside - UDP					
Port	Bad %	Bytes Sent From	Port	Bad %	Bytes Received On
31518	100.00%	7,517,177,675	10291	99.74%	3,253,848,069
26782	99.98%	7,709,548,264	16393	90.53%	3,722,586,005
16393	91.14%	8,698,919,312	27005	99.92%	5,533,401,186
16402	100.00%	13,497,964,368	16402	99.99%	9,393,413,986
64916	97.61%	16,617,017,804	64916	95.55%	76,851,256,026

are not being taken into consideration and with further analysis, it may turn out that the majority of the bad conversations in our sample have relatively low scores.

Table 5.7 presents the hosts on the outside of the network (i.e. the Internet) with the largest bad traffic volumes. The Bad column has the same meaning as with the previous table. The traffic volume is calculated by summing the bytes of data sent to a host or received from a host by the internal network. Significantly, most of the hosts have 100% bad traffic with the UDP traffic volumes all being part of conversations from our bad sample. Of all the hosts, 104.44.200.136 appears to be the most suspicious with traffic flows accounting for 14.7% and 6.2% of bytes sent out to or received from the Internet. As mentioned above, on investigation, this was found to be Skype-related traffic.

Table 5.9: Top bad traffic sources and destinations by outside port

Top Ports Outside - TCP					
Port	Bad %	Bytes Sent To	Port	Bad %	Bytes Received From
20	100.00%	2,168,447,592	50068	99.98%	5,648,897,809
6881	99.16%	2,183,768,801	6792	100.00%	5,666,005,400
22	100.00%	5,063,617,458	8081	99.88%	7,270,105,495
1433	99.95%	5,303,350,629	8182	100.00%	8,624,271,880
514	99.99%	18,519,726,767	8777	99.99%	80,776,292,411
Top Ports Outside - UDP					
Port	Bad %	Bytes Sent To	Port	Bad %	Bytes Received From
26861	100.00%	7,517,177,991	28743	100.00%	4,155,673,218
16402	100.00%	7,662,155,068	16402	100.00%	6,205,749,627
4500	97.66%	16,848,356,546	23635	100.00%	8,660,542,260
3480	100.00%	41,425,161,441	3480	100.00%	25,601,162,842
3478	87.50%	75,084,863,472	4500	95.59%	77,339,315,707

Table 5.8 presents the ports on the internal network with the top bad traffic volumes. As before, the Bad column represents the percentage of traffic volume that our bad sample contributes as part of the total traffic for that port. The traffic direction is classed as volume sent to the Internet and volume received from the Internet respectively. TCP port 45554 immediately stands out with the outbound traffic flow accounting for 12% of the outbound traffic in the bad sample. The position of this port at the top of the sent and received lists is a strong indication that there is a service hosted on this port inside the network. In the UDP list, the statistics for port 64916 have a similar pattern to the top value in the TCP list. In this case, however, the direction of the biggest flow is reversed with the inbound traffic accounting for 14.7% of total.

Table 5.9 lists the ports on the outside (i.e. Internet) and their bad flow data volumes. For TCP, traffic flows to port 8777 on the outside account for 15.6% of outbound traffic, 10 times the next closest entry. The outgoing TCP traffic from port 514 is also of concern, as this port is typically used by insecure services such as rlogin, rsh etc. The top scorers on the UDP section also account for significantly more traffic than the next closest values. Port 3480 traffic stands out, as it accounts for a large amount of both outbound and inbound traffic (8.0% and 4.9% of the total traffic in our bad data set) and possibly represents a hosted service on the Internet.

In this analysis, we have only examined traffic in terms of bytes sent and received and it is only a subset of what could be accomplished. A similar exercise can be carried out looking at bad conversations in terms of total packet counts, bytes per flows, bytes per packet, bytes per conversation, etc. In our review of the data, we also only looked at a

limited number of data points of each type due to resource constraints, but there is scope for a much more detailed analysis of the data presented so far. Based on this review we can already highlight the following items for further investigation:

1. Increase in bad UDP traffic halfway through the sample period.
2. The spike in incoming bad TCP traffic on the 14th of January.
3. The spike in bad TCP traffic on the 25th of February.
4. Large UDP traffic flows from 192.0.2.130.
5. Large UDP traffic flows to 104.44.200.136.
6. An unknown service being hosted on port 45554 on an inside server.
7. A large UDP inflow to port 64916.
8. Large TCP traffic flows from port 8777 on the outside network.
9. Large outbound TCP traffic flows from port 514 (may be insecure services).
10. The large UDP traffic flows to and from port 3480 outside the network.

Items 1,2,3 and 5 are examined in more detail in section 5.4. Item 4 was discussed above and was determined to be Skype traffic.

5.3.4 Score Statistics

In this section, we analyse the scoring characteristics of our bad sample data. This is done by looking at the score of the conversations and correlating this with hosts and ports. We also look at score distribution and high scorers. We will not be comparing the sample data back to the complete scored data set, as that contains variations on good scores, bad scores and no scores which will not make for a valid comparison.

Table 5.10: Scored flow statistics

Statistic	Value
Minimum score	10.0
Average score	103.5
Maximum score	410.0
Median score	230.0

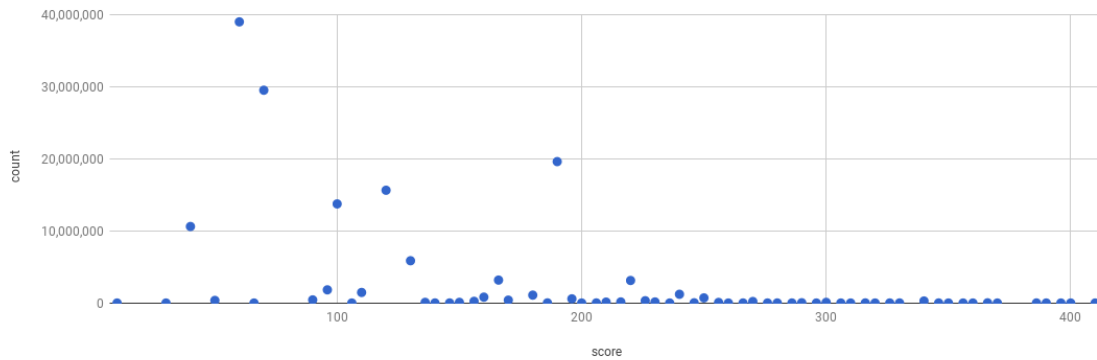


Figure 5.15: Bad sample score distribution

Looking at the scored flows in our sample, we can calculate some simple statistics as shown in table 5.10. Plotting the distribution of scores shows a long tail effect as expected with a significant number of low scored conversations and a low number achieving high scores. Figure 5.15 shows the distribution of scores in our sample and figure 5.16 zooms in on the tail showing the distribution of scores above the median value of 230. We can make a number of observations from these charts:

- The most common score is 60 (38,979,615 records) - this value matches the score assigned to possible scans and the conversations in this data point are most likely scan candidates.
- In the overall sample, charted in figure 5.15, there are unusual peaks at a score of 120 and 190 (15,635,120 and 19,606,244 conversations respectively).
- In the second half of the sample, shown in figure 5.16, there is an extraordinary peak of just over 279,899 conversations with a score of 340 and another one of 24,579 conversations with a score of 366.
- Only 154 conversations were scored with the maximum score of 410.

In figure 5.17, the sum of all bad scores in our sample is plotted by day. This immediately brings out a significant peak on the 24th of January. This peak is not isolated on the

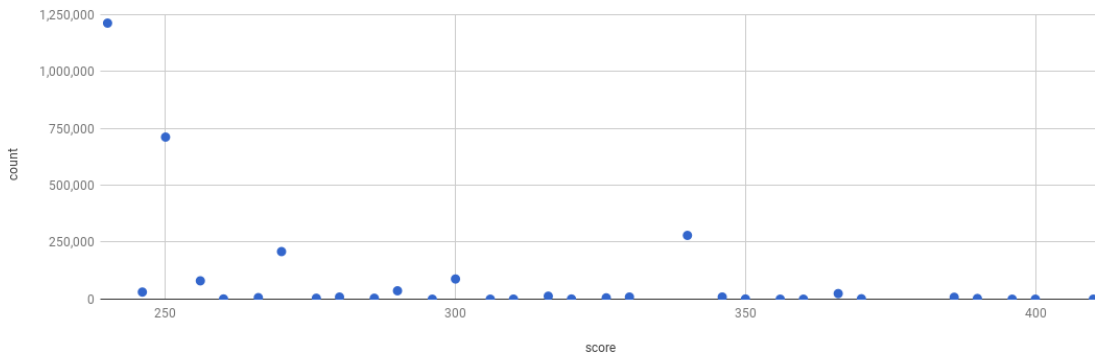


Figure 5.16: Bad sample score distribution - expanded tail

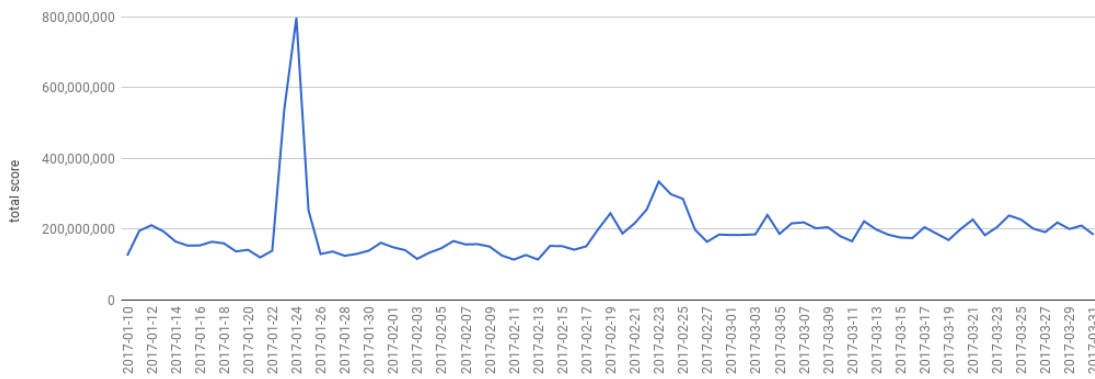


Figure 5.17: Total score by day

24th of January, but rises on the 23rd and tails off on the 25th. A smaller but longer peak of bad score total is visible around the 23rd of February. Plotting the average scores for all conversations, TCP only conversations and UDP only conversations in figure 5.18 shows an anomaly around the 24rd of January. In the chart, a significant drop in total and TCP average score can be seen on the 24th January. The drop in average score at the same time as an increase in total scores is most likely caused by a surge in traffic that is scored by a single bolt resulting in a large number of low scored records. This can be confirmed through further analysis by drilling down into the details of the scoring on the day in question. A final point to note is that the TCP daily average closely follows the total daily average, indicating that TCP scoring makes up most of the bad scores.

In summary, a brief look at some of the statistics of the scores has raised a number of data points that require further investigation:

1. Investigate a sample of the 154 conversations with the highest score.

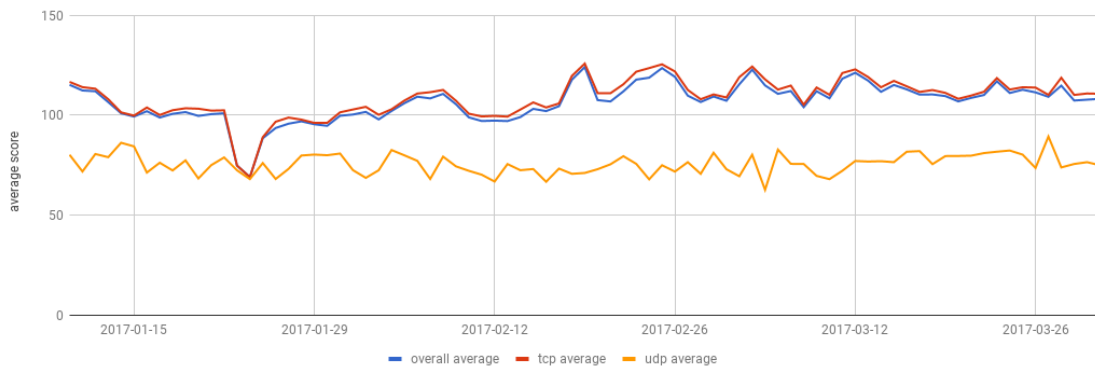


Figure 5.18: Average score by day

2. Investigate a sample of the peaks identified in the second half of the score distribution (figure 5.16).
3. Carry out a more detailed analysis of the traffic on the 24th of January, focusing on the possible surge of a particular traffic pattern.
4. Examine the small surge in total bad traffic score around the 23rd of February.

Items 1,3 and 4 are examined in more detail in section 5.4.

5.3.5 Tag Statistics



Figure 5.19: Tag count per day

In the final section of the review of our processing output, we consider the tags that have been applied to our sample. The chart in figure 5.19 shows the total tag counts assigned to conversations per day. While mostly a constant value, there are two dates that stand

out - the 24th January and the 23rd of February. Both are dates that have been observed as containing points of interest in previous sections in this chapter (see sections 5.3.3 and 5.3.4).

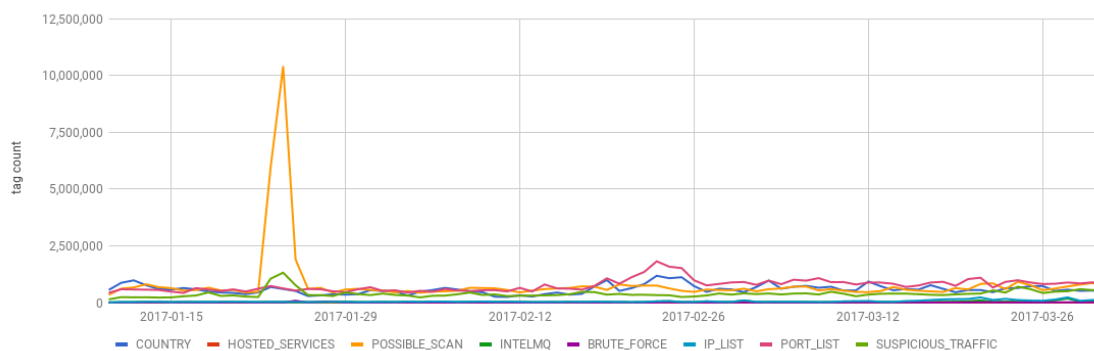


Figure 5.20: Tag count per day broken down by tag

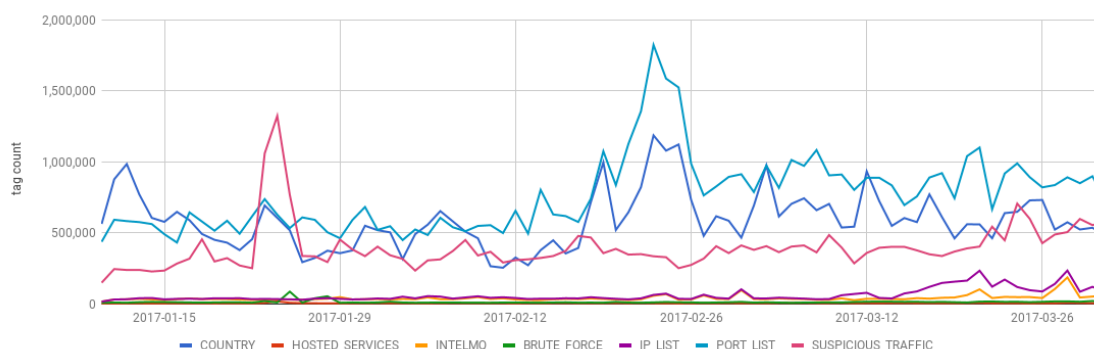


Figure 5.21: Tag count per day broken down by tag (scans excluded)

In figure 5.20 the tag categories are broken out and the individual counts of tags assigned to conversations plotted. This allows us to see the details around some of the spikes observed earlier in this section. The spike in January appears to be a scan related peak and after some investigation, the one in February was found to be linked to an increase in traffic from a combination of blacklisted ports and countries. By removing the scan data from the chart we can see more detail on the other tags in figure 5.21. The first thing we can observe is there is an increase in traffic dark IP correlating to the scan on the 24th of January, indicating that it may have been a network wide scan (of the internal network). A high number of conversations that were scored for potential scans only on this day

would have resulted in a peak of total score on the day, but with a corresponding drop in average score. This peak in the total score, along with the drop in average score, can be seen in figures 5.17 and 5.18. The peak of blacklisted port traffic in February is matched by a peak in traffic from a blacklisted country, indicating that a traffic flow has taken place from a single external source, possibly on a limited number of ports. The network traffic correlating to this event can be seen in figure 5.12. This event will be examined in more detail in the next section.

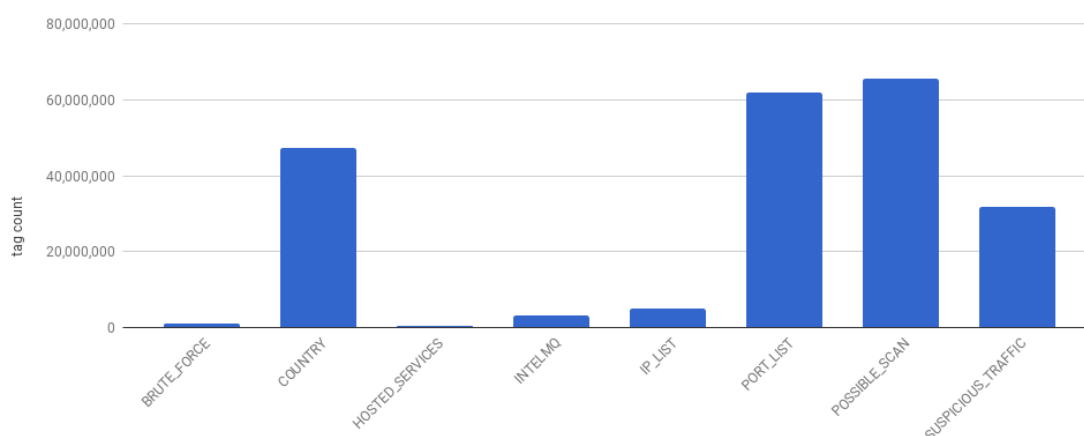


Figure 5.22: Tag category breakdown

In figure 5.22, the number of occurrences of each category is charted. Suspected scans and unknown port traffic tops the list with dark IP traffic and blacklisted country traffic also featuring a significant number of times. Of interest is the fact that 31.4% of all our bad traffic sample originates from our two blacklisted countries (Russia and China as defined in section 5.1.3). If we remove data points that have only been flagged as bad due to their origination from one of these blacklisted countries, we find that we still have 20% of the bad conversations in our sample originating from said blacklist. In other words, 20% of bad conversations with more than one tag involve Russian or Chinese hosts.

Figure 5.23 shows the distribution of tags per conversation. The majority of conversations (64.2% of total) have a single tag and 94.5% of all conversation have either one or two tags. This indicates that the vast majority of the traffic conversations in our bad sample represent potentially minor threats when viewed individually. The remaining conversations have more tags and a resulting higher score, indicating more likelihood of a threat. The highest number of tags is six per conversation with 36,269 conversations in

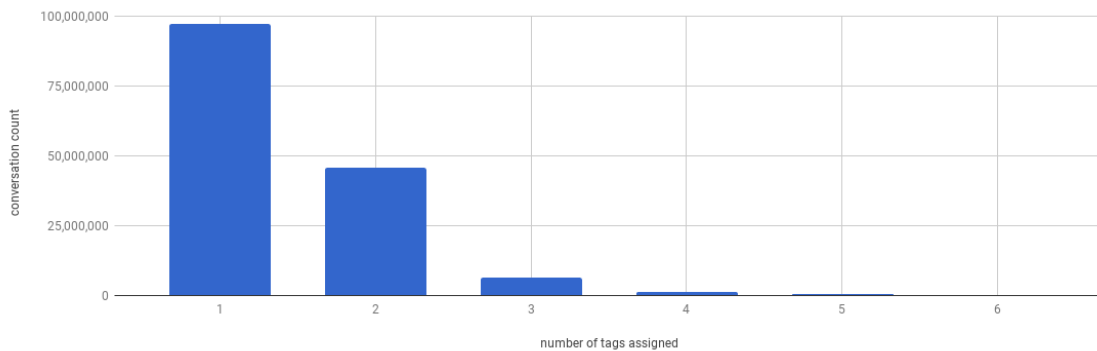


Figure 5.23: Tag count per conversation distribution

this group. The conversations with four, five and six tags each represent only 1.7% of the total bad sample, but still account for over 1.9 million conversations.

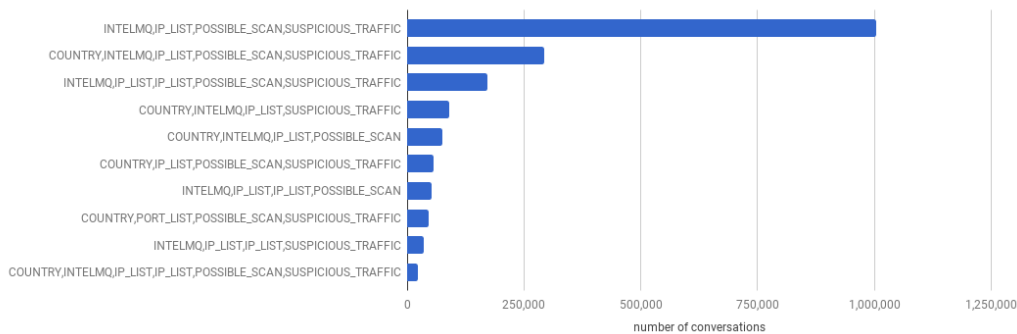


Figure 5.24: Tag combination counts (4 tags or more)

Up to now we have looked at the statistics around how many tags were assigned. In order to get more insight, we can look at the tag combinations assigned to conversations and the permutations thereof. By this we mean the actual set of tags that conversations have been assigned. For purposes of this work, we will not examine conversations with a small number of tags (and correspondingly a small bad score). It is not that these conversations are unimportant, but they require significant effort in terms of interpretation. In figure 5.24, we chart the top ten tag category combinations for conversations that scored four tags or more (the top 1.7% from figure 5.23). The most common combination are 1,004,541 conversations with the combined tags “*INTELMQ*, *IP_LIST*, *POSSIBLE_SCAN*, and *SUSPICIOUS_TRAFFIC*”. This indicates potential scan traffic from hosts that are listed in one of our static blacklists and in the IntelMQ threat intelligence data sources. The traffic was also tagged for being directed at dark IP ranges. Drilling down into the conversations with the most tags (six each) we can see the details in fig-

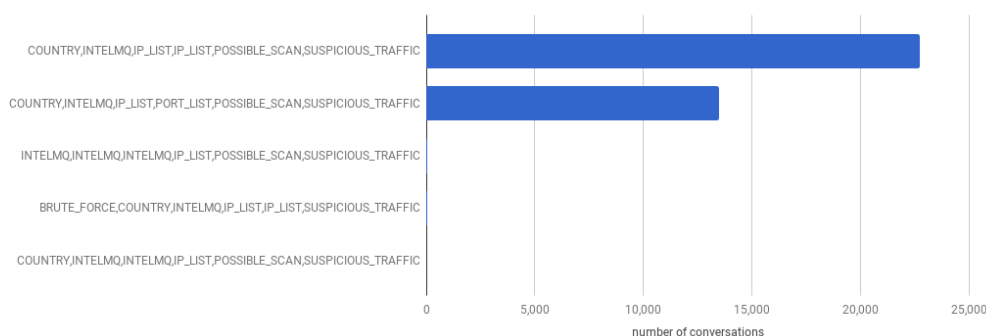


Figure 5.25: Top tagged conversation permutations

Figure 5.25. From the combinations it appears that conversations linked to scanning from blacklisted IP hosts are being flagged with the most tags.

In this section the tagging outputs were presented and reviewed. In a number of cases the results correlate with data points of interest highlighted in previous sections and support further investigation (24th of January and the 23rd February). There is a significant number of conversations that were flagged with six tags. Of these, two permutations make up the majority and could possibly be related to a small number of sources that may be identified through further analysis.

5.4 Analysis of Results

In this section, a more detailed investigation is presented of the initial findings from section 5.3. In each sub section, we will examine one of the suspicious data points identified earlier in this chapter. The analysis was done using a combination of database queries and data interrogation using Kibana and Elasticsearch (as described in 4.10). In the final part of this section, we look at some of the scoring approaches and discuss the effectiveness.

5.4.1 Increase in Bad UDP Traffic

In figure 5.13, an increase in bad UDP traffic is observed midway through the sample period. Using Kibana to look at UDP traffic only, it was noted that for this period conversations to host 163.172.215.161 accounted for the majority of the bad traffic in this period by host. A small amount of traffic was observed on the 9th of February, but

it really only picked up from the 19th and from then on was sustained for the rest of the period. Filtering our data using this IP, it was further discovered that the traffic comprised 142,433 conversations from port 5060 on the inside of our network to ports ranging from 5060 to 5107 on the remote host. The traffic was evenly spread along most hosts (live and dark) on the inside network with an average of 708 conversations per host.

The remote host is located in Amsterdam and is part of a netblock assigned to a European ISP (Online SAS NL). The host in question has the DNS entry 163-172-215-161.rev.poneytelecom.eu. Port 5060 is assigned to SIP traffic and the nature of this traffic appears to be scanning for open PBX services. There have been a number of reports of similar traffic from this host in the same time period (see Abuse IP Database (2017) for an example).

5.4.2 TCP Traffic Spike - 24th of January

On the 24th of January a spike can be seen in the bad TCP traffic (see figure 5.14) in our sample. This peak began on the 23rd and tailed off on the 25th. Looking at the period in question in more detail, the following was observed:

- The majority of the bad TCP conversations on the day involved the hosts 54.175.86.120 and 54.173.194.115.
- The two hosts in question are part of the Amazon AWS infrastructure.
- Together these hosts accounted for 9,494,245 bad conversations over the period.
- The traffic in question was flagged with a range of tags, but the majority was by far classified as scanning attempts.
- Traffic from these hosts came from a small range of high-value ports, but was directed across the entire range of hosts and ports on the internal network.

Investigating further it was determined that the two hosts are part of a range used by a security company for scanning their clients networks. The company in question, Paladion, lists these ranges on their website for whitelisting purposes¹. This leads us to conclude that the traffic in question was possibly scanning done as part of a purchased service.

¹<https://paladion.net/customers-ip/>

5.4.3 Spike in Activity - 23rd to 25th of February

From the 23rd of February there was a spike detected in suspicious activity. The first part was an increase in bad conversations (in terms of score and tags) and the second part (up to the 25th) constituted an increase in bad conversation traffic volumes. On further investigation, it was found that a number of unusual events took place over this period, including amongst others the following:

- Suspicious Server on internal network. From other flagged traffic it was determined that an unknown service was running on two internal hosts: 192.0.2.189 and 192.0.2.190. This service is hosted on port 45554 and appears to be a SOCKS proxy. This theory was supported by the presence of these IP addresses in historical listings on a website known for publishing open proxy addresses². Traffic to and from this proxy has been noted throughout the sample, however, in the period in question it surged, accounting for 64.6% of all bad conversations.
- On the 24th of February a large volume of outbound UDP traffic was observed from port 49952 on 192.0.2.130. This traffic accounted for 56.1% of all outbound traffic for the day and was to a single port 26530/UDP on host 169.239.44.14 (located in Durban and belonging to and ISP called iSPOT). The traffic flow lasted six hours in total and ran from 11pm on the 23rd to 6am on the 24th. No other traffic was observed to this host in this period. There is little more information available, other than this IP has been reported before for alleged abuse³.
- A network scan was picked up from a MWeb based IP, located in Johannesburg. The scan in question evenly targeted a small number of well-known ports, including FTP, SSH, HTTP, POP, IMAP and SMTP across 42 hosts. The probe was unusual in nature in that the source was repeatedly probing the same set of ports every minute for hours at a time, even when there was no response.
- On the 25th 12GB of TCP traffic was sent out of the network from host 192.0.2.167 to port 8777 on a host with IP 185.163.109.110 located in Bucharest Romania. Further investigation indicated that the site was part of the thevideo.me video upload network and this outflow appears to be a video upload and thus a legitimate traffic flow.

²<http://ssh-dailyupdate.blogspot.co.za/>

³<https://www.abuseipdb.com/check/169.239.44.14>

A number of big outflows of data appear at various places in the sample. In most cases these appear to be innocent, but in a more secure environment, these could be indicators of data exfiltration in this context. While the geographic locations of hosts has been referenced a number of times in this section, it should be borne in mind that the accuracy of IP geolocation lookups has limitations (see section 4.4.3 for more detail).

5.4.4 Top Scoring Conversations

Twenty three remote hosts appear in the 154 top scoring conversations. Of these, three Chinese hosts account for 76% of the top scoring conversations. All three hosts appear on both AlienVault blacklists and in the IntelMQ threat database. The entries in the IntelMQ database are listed with the reason “malware : malicious code”. A small number of ports are targeted by these hosts, the most notable being SSH (port 22/TCP), Microsoft Directory Services (port 445/TCP) and Microsoft Remote Desktop (port 3389/TCP). Unusual ports that are scanned are 7547 and 2222. The former is associated with a DSL modem vulnerability targeted by the Mirai botnet⁴ and the latter is notable as being associated with a rootshell left by the AMD buffer overflow exploit⁵.

Of special interest are numerous short conversations from two of the hosts (122.96.140.226 and 183.62.7.34) to port 3389 on 192.0.2.134. This port is normally used for remote desktop access on Microsoft Windows machines. The numerous short conversations are likely a sign of brute force login attempts.

5.4.5 Top Tagged Conversations

In our bad conversations sample 36,269 records were flagged with a top total of 6 scores. Of these 84.3% came from five Russian hosts, of which four were in a contiguous netblock giving a strong indication that they were part of the same organisation. All conversations in this group were identified by the bolts using the IntelMQ and AlienVault threat intelligence, reflecting widespread presence of these hosts in threat intelligence databases. In the cases of the IntelMQ tags, the reasons for being listed were given as “malware : malicious code”. Broadening the analysis revealed that all five hosts were involved in 141,203 bad conversations in total.

⁴<https://securityintelligence.com/mirai-evolving-new-attack-reveals-use-of-port-7547/>

⁵<http://galaxy.cs.lamar.edu/~bsun/forensics/slides/Exploits.pdf>

The traffic patterns from the single host (191.96.249.97) appeared to be targeting services on port 80 and 8080 with some of the traffic matching the HTTP brute force heuristics implemented in the topology (see section 5.4.6). The traffic from the group of four (95.213.177.123 to 95.213.177.126) was a bit more varied. Some of it was traffic to the suspected SOCKS proxy on port 45554 mentioned in section 5.4.3, some was traffic scanning various internal hosts on port 8080 and finally there were 22 conversations from host 192.0.2.189 on the inside network to port 80 on 95.213.177.124. In light of the malware classification from the IntelMQ data, this outbound traffic could be considered as possible command and control traffic.

5.4.6 Bolt Efficacy

The outputs are a consequence of the scoring approaches implemented in the topology bolts. These outputs are only as effective as the strategies used by the bolts and these strategies require some reflection. During the analysis of the output it was realised that not all tags are equal - depending on the nature of the tag and the source of the information, some are stronger indicators than others. While this can be represented in the comparative scores that are assigned, it must be kept in mind when looking at the actual outputs during analysis.

In figure 5.22, we can see a comparison of the number of times each category of tag is used to flag a conversation in our sample. In top position is the tag for potential scans. In our sample, 43% of the conversations had a likelihood of being part of scan data. A single potential scan conversation is in and of itself not enough to raise concern. The effectiveness of this tag relies on detecting a significant number of these conversations with some common attributes (e.g. same source or same target port, see section 4.7.3 for more information). Similarly for the tag in position number three, traffic flagged as potentially suspicious because it was directed at allocated or dark IP address space. By itself, such a flagged conversation is not necessarily a potential threat, but when seen in context or on in conjunction with other tags the potential of a problem is increased. The above also applies to the suspicious country tag - it is a hint of a problem, rather than a strong indicator and has to be analysed in context. Further to this, it could be argued that the tags in this class should be assigned lower scores.

The next class of tags that require consideration are those that score based on traffic to or from designated ports. One group uses a list of “known” ports or services as its base,

while the other relies on foreknowledge of what traffic is expected in a network. The latter class (the `PORT_LIST` tags) was the number two assigned tag category with the majority of the traffic in this group being classified as unknown TCP traffic. In the second class, the `HOSTED_SERVICES` category barely featured. In both cases the scoring was not as effective as it could have been due to a limited understanding of what traffic was expected and what was not. The inside network appeared to have limited controls in terms of what was permitted or not and as a result, there was a lot of potentially suspicious traffic that was most likely sanctioned traffic. The scores assigned to this class should be dependent on the strength of the foreknowledge. The more understanding there is of what is or is not allowed, the higher the scores and vice versa.

The blacklist and threat intelligence tags appeared to be relatively effective with most of the significant issues found having tags from one or more of these sources. There was, however, a lot of overlap between one of the static lists and the IntelMQ dynamic threat feed with the latter proving to be the most comprehensive. Going forward it would probably be enough to include the IntelMQ bolt only but using a higher score for the matches. The source of the data, the IntelMQ application, can also be further configured to broaden its data sources.

The final scoring bolt that was considered for effectiveness is one of the examples that was taken from prior work. The SSH brute force detection heuristic from Hofstede and Sperotto (2014) and discussed in section . Reviewing the conversations identified by the bolt it proved to be very effective. In one example, a single host from Shenzhen China was observed attempting SSH logins on four internal hosts. Of the 31,598 SSH conversations in total, 27,399 or 86.7% were identified and tagged as brute force attempts. This gives strength to a view that using more tried-and-tested approaches may be the way to go, leveraging the work of others in a combined approach for threat detection.

The final dataset, while reduced in size, was significantly richer in information. This raised its own challenges when trying to look for patterns and indicators of potential threats. The interactions and relationships between the different scored conversations are complex and require a lot of work to trace and map out. A review of the scoring in terms of what strategies to use and the associated scores may help to solve this problem, but the actual handling of the final outputs also requires more work.

5.5 Summary

In this chapter we looked at our data, the processing of the data, the raw outputs and then provided a brief analysis of a sample of the output. In terms of our goals, we have established that on commodity hardware we are able to process a significant volume of NetFlow data with relative ease. Our sample of 1.4 billion rows was easily processed in under 24 hours, indicating that high throughput can be achieved. In terms of latency, we saw processing times of under two seconds, but there is evidence from the scaling tests that this could be reduced significantly. In addition, the testing indicated that more work can be done on increasing performance.

The scoring process reduced the size of the data we need to look at by 50% at least. Further filtering of this data using score attributes reduced our data set of interest to 10% of the original amount. The reduced data set, however, was vastly improved in terms of quality of information, which in turn produced its own challenges. Analysis of this data indicated that the scoring and tagging was definitely able to identify potential issues in the data. It also, however, raised some challenges in terms of interpretation of the data and extraction of value which both require further investigation and research. In solving one problem, another challenge may have been created.

Chapter 6

Conclusion

In the final chapter of this thesis, the work presented is summarised (section 6.1) then reviewed and evaluated against the original objectives (section 6.2). In section 6.3, the significance of the developed framework and the implications of the outcomes are discussed. Finally, in section 6.4 potential future work is discussed covering improvements to the framework, enhancements to the scoring and other potential applications of Themis.

6.1 Summary

This document describes a proposed framework for real-time security incident detection using only NetFlow data. The work is presented in six chapters taking the reader through the journey from inception to findings. The first two chapters cover the work proposed along with context and background information. In chapter 1 the framework concept is presented as a possible solution to real-time threat detection using flow data. The challenges of the problem are presented and the goals and scope of the project defined with respect to these challenges. Chapter 2 consists of two parts: the first part covers some technical background on the various technologies and concepts around which the work is centred, while the second part presents a literature survey which covers prior work around incident detection approaches with NetFlow and Big Data and scoring. Finally, a selection of open source systems that are similar in nature are presented.

Chapters 3 and 4 focus on the architecture, design and implementation of Themis. Architectural considerations are first discussed, followed by the system requirements in

terms of scoring and scaling. Where applicable, different options for technology choices are given and the choices are justified within the context of the goals of the project. The implementation of Themis is documented in detail and covers all aspects from the tools and technologies, languages used and data flow to the implementation of the individual scoring components.

The results and findings chapter begins with a summary of the data sample used and details on the configuration of the system used for processing. The data processing throughput and latency statistics are given with some background on the work done to optimise these numbers and uncover bottlenecks. The outputs of the processing run are examined from various angles looking at the traffic statistics, scoring and tagging outputs in order to identify potential incidents. The chapter finishes off looking at a selected number of these events in detail in order to evaluate the framework's ability to identify anomalies. The final chapter is a summary of the work done, an evaluation of the achievements and a discussion of extensions to Themis and possible future work.

6.2 Research Evaluation

In chapter 1, the two primary goals of the project are specified as expanding on the flow scoring approach and then the implementation of a framework for real-time flow scoring.

The achievement of the *first goal* is evident in the variety and number of scoring bolts implemented as described in chapter 4 and specifically listed in table 4.1 in section 4.9. In the original work scoring was done in a manual, iterative, batch processing mode using nine different scoring tests. In this work, the scoring methodologies are clearly defined and implemented as standalone, independent components each capable of working in isolation. Eleven different scoring components are implemented, some with dedicated scoring functionality but many are configurable allowing a broader range of anomalies to be targeted. The range of scoring options configured during testing was set at sixteen, ranging from static lists through to heuristics adopted from other research. In addition, the enrichment process was improved and additional features such as flow matching added. In section 5.3 the scored outputs from the processing are presented and a significant number of anomalies easily identified. Investigating a selection of these further in section 5.4 uncovered a number of potentially harmful activities in the network traffic. The ease

with which these incidents were identified and the richness of information available for further analysis is a strong indicator of the effectiveness of the scoring approach.

Taking these components and implementing a real-time scoring platform was the *second goal* of this work. In section 1.1 we specify theoretical expected throughputs in terms of flows per second for different link speeds. In this section we also discuss the criteria for real-time incident detection in terms of processing latency expectations. In section 5.2 the throughput and latency statistics for flow processing on a single 4 core node are presented. The system was able to handle a throughput of 18,800 records per second with an average latency of 1.961 seconds per record. Taking the metrics discussed in section 1.1 into account, this latency falls well within the generally accepted value of real or near real-time. Using table 1.2 as a basis, Themis can process the flows generated by a 360 Mbps link running at an average of 50% load. While many backbone links run at speeds and loads far exceeding this, it must be remembered that only a single processing node was used in this project. In summary, Themis can in theory do scoring of NetFlow data in near real-time for link speeds of up to 360 MBps using similar hardware.

The final overarching achievement of this work is that the system built is, as intended, an extensible framework. The different scoring bolts are configurable, the data sources used are configurable, the scores and tags are configurable and finally, and most importantly, the scoring topology can be changed as required. In section 5.1.6, different topologies were configured for performance testing purposes. In most cases these were simple configurations of only two or three bolts but this was not a limitation of Themis and a number of permutations were tried. This can easily be extended to production deployments where administrators can choose to deploy simpler or more complex topologies as required. This feature above all opens the possibility of using the framework in many different environments adapting to the situational requirements as needed. Finally, it should be noted that adding new components to the system requires the implementation of a new bolt, a simple and straightforward process using a standard Java code template opening the way to extending Themis with new scoring or enrichment components.

6.3 Significance of Research

The work conducted has produced a number of outputs of significance. The first of these is the expansion on the NetFlow scoring research. Using scoring as a primary means of

classifying flow records for filtering or identification purposes, as is done in this work, is a new approach to anomaly identification in NetFlow data. The effectiveness of this still requires more investigation to verify and validate, but the findings presented in chapter 5 show promising results. Taking this work further and showing that real-time scoring is possible adds to the significance of the work and demonstrates that the application has real-world potential. The use of Big Data architectures and technologies as a foundation for implementation has shown that scaling to large data loads is possible.

The implementation of this work as a framework provides a basis for extensions and/or further experimentation with this approach. This is significant as the nature of a framework is such that it allows for changes or extensions to its functioning. The work serves as a basis on which others can build or as a system that can be integrated into for complementary processing purposes. The use of this framework as a foundation will hopefully serve to bootstrap future research into real-time event detection.

6.4 Future Work

The focus of this work was the design and implementation of the Themis framework which limited the scope of the work in terms of breadth of threat identification. Additionally, during the course of the implementation and testing a number of shortcomings and extension points were identified. And finally, when reviewing the outputs and findings a number of changes and enhancements were identified. All of these give rise to a range of future work opportunities.

6.4.1 Enhancing the Scoring

Where scoring is concerned there are many candidate areas for future work. In section 5.4.6, the effectiveness of the different bolts was discussed and compared. As noted, some of the bolts performed better than others. More work is needed on evaluating each bolt's effectiveness. This work can be done by using previously labelled samples such as the CTU data sets (García and Uhler, 2017), and measuring the effectiveness of the bolts in detecting aberrations. The second area that requires attention is expanding the capabilities of the scoring bolts. The ones implemented in this exercise serve as a good cross section of what is possible, but are by no means comprehensive. Further to this,

adding more enrichment bolts (such as WHOIS data), should be considered. Expanding out the threat intelligence sources can also be done using a wider range of suppliers. One option to be considered is whether or not threat intelligence data should be added as enrichment first and then used as scoring instead of simply scoring. The enrichment information could prove more useful for more complex scoring and in the output and analysis.

Opportunities exist for leveraging the work of other researchers much in the same way as described in section 5.4.6. In this project, two examples of prior work were implemented with reasonable success (see discussion in section 5.4.6). For the system to be truly effective as a framework, the addition of further scoring components based on proven research should be exploited for maximum gain. A final extension to be considered is a feedback loop whereby data from previously scored flows can be used as a scoring mechanism. For example, hosts that appear multiple times in highly scored flows may be flagged accordingly.

6.4.2 Candidate Scan Processing

One of the bolts in the implementation identifies flows that are potential scan candidates (section 4.7.3). Currently these flows are sent out to a Kafka queue for further analysis but further analysis was deemed to be outside the scope of this work due to time constraints. Work is required to implement further processing on these flows with an emphasis on applying classifications such as those presented in Barnett and Irwin (2008). Another aspect to consider is expanding on the identification of traffic to dark IP addresses for advanced threat detection (see Fachkha and Debbabi (2016) for examples of possible work). The nature of the processing is more batch oriented as scanning detection requires looking for similar flows over a time period. Approaches such as map-reduce over a time frame of these records would in all likelihood be the means of confirming scanning.

6.4.3 Performance Improvements

In section 5.1.6, the throughput and latency metrics of Themis are presented. While the overall performance was deemed to be acceptable, a number of issues were noted with the throughput limitations caused by some of the bolts that appeared to be introducing bottlenecks in the processing stream. More work is required to understand these limitations

and to build improvements to the implementation. Investigating the caching of scored flows and doing quick lookups before fully processing each record may serve to reduce load and improve throughput and latency.

6.4.4 Scaling and Elasticity

The testing of Themis was done on a single node and the scaling out across multiple nodes has been assumed as possible due to the capabilities of the technologies chosen. However, in reality things are never as simple as that. There is opportunity for experimentation with the multi-node scaling of the framework using these features of Kafka and Storm. Included in this work is testing of the elasticity of the overall solution - can it dynamically scale up and down and how does it perform when doing so? The use case for this would be to extend Themis to allow for scaling up or down on demand.

6.4.5 Other Streaming Technologies

Apache Storm was the stream processing technology used for this work. The implementation of the framework is not, however, tightly coupled to the use of Storm. There are a number of other stream processing frameworks that could be evaluated for the same purpose. The most obvious choice is Apache Spark which is the main competitor to Storm. There are, however, stream processing engines that are not Java/JVM based that may offer similar or better performance coupled with easier implementation requirements (for example Wallaroo¹, a pure Python streaming framework).

6.4.6 Analysis of Output

For this research, the majority of the output was processed by hand or examined using adhoc dashboards in Kibana. There is significant scope for work on building out more robust visualisation and alerting tools using the output of Themis. In particular, a focus on identifying major anomalies and allowing for easy drilling down into them is required.

¹<https://www.wallaroolabs.com>

6.4.7 Production Testing

Themis has been tested in a controlled environment with real-world data. More testing is required and specifically production deployment testing. This would require getting a live NetFlow feed from an operational network, configuring the framework for this network and then running the system for an extended period of time with real data.

6.4.8 Machine Learning Opportunities

There are at least two opportunities for introducing machine learning (ML) into Themis. The first is to look at using the technology as part of the scoring. There is extensive work on the use of ML in threat detection and this can be leveraged as a new type of scoring bolt. The second potential use of ML is in the analysis of the output data. Through the application of unsupervised learning more insight may be gained from the scored flows.

6.4.9 Network Traffic Accounting and Scoring

While this work was focused on the security aspect of NetFlow processing, this framework also lends itself to simple network traffic processing and even scoring. The implemented utility bolts could be used for simple network traffic analysis and the scoring could be extended to allow for traffic categorisation.

6.4.10 Other Uses

Real-time scoring using streaming for security event detection can be applied to other data sources. For example, this approach can be used on access logs, browsing records, host activity, etc. Additionally, Themis could be extended to ingest multiple sources of data with a view to correlating the data in real time. For example, user access logs and NetFlow data can be compared in order to pick up unauthorised or unknown activity from an unattended machine.

References

- Abuse IP Database.** AbuseIPDB 163.172.215.161. <https://www.abuseipdb.com/check/163.172.215.161>, 2017. [Accessed: 2017-11-01].
- Agilent Technologies.** Mixed Packet Size Throughput. Technical report, 2001.
- Akida, T.** The world beyond batch: Streaming 101. <https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101>, 2015. [Accessed: 2017-09-23].
- Amini, P., Azmi, R., and Araghizadeh, M.** Botnet Detection using NetFlow and Clustering. *Advances in Computer Science: an International Journal*, 3(2):139–149, 2014.
- Apache Organisation.** Apache Metron: Real-Time Big Data Security. <http://metron.apache.org/>, 2016. [Accessed: 2017-11-15].
- Apache Software Foundation.** HowScoresAreAssigned - Spamassassin Wiki. <https://wiki.apache.org/spamassassin/HowScoresAreAssigned>, 2009. [Accessed: 2017-09-26].
- Apache Software Foundation.** Event Time / Processing Time / Ingestion Time. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_time.html, 2017. [Accessed: 2017-11-18].
- Arbor Networks.** Packetpig - Open Source Big Data Security Analysis. <https://www.arbornetworks.com/blog/asert/packetpig-open-source-big-data-security-analysis/>. [Accessed: 2017-11-15].
- Arkko, J., Cotton, M., and Vegoda, L.** RFC5737 IPv4 Address Blocks Reserved for Documentation. [https://tools.ietf.org/html/Internet Engineering Task Force \(IETF\)](https://tools.ietf.org/html/Internet%20Engineering%20Task%20Force%20(IETF)), 2010. [Accessed: 2017-10-01].

- Bar, A., Finamore, A., Casas, P., Golab, L., and Mellia, M.** Large-scale network traffic monitoring with DBStream, a system for rolling big data analysis. *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, pages 165–170, 2015. doi:10.1109/BigData.2014.7004227.
- Barnett, R. J. and Irwin, B.** Towards a Taxonomy of Network Scanning Techniques. *The 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries riding the wave of technology riding the wave of technology - SAICSIT 08*, pages 1–7, 2008.
- Buhl, H. U., Röglinger, M., Moser, F., and Heidemann, J.** Big data: A fashionable topic with(out) sustainable relevance for research and practice? 2013. doi: 10.1007/s12599-013-0249-5.
- Callaghan, D.** Big Fast Data Perishable Insight At Scale. <http://www.sparksignite.net/index.php/2015/06/14/big-fast-data/>, 2015. [Accessed: 2017-11-23].
- Čermák, M., Tovarák, D., Laštovička, M., and Čeleda, P.** A performance benchmark for NetFlow data analysis on distributed stream processing systems. *Proceedings of the NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, (Noms):919–924, 2016. doi:10.1109/NOMS.2016.7502926.
- Chandrashekar, P., Dara, S., and Muralidhara, V.** Feasibility Study of Port Scan Detection on Encrypted Data. *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, (November):109–112, 2015. doi: 10.1109/CCEM.2015.18.
- Choudhary, S.** Usage of Netflow in Security and Monitoring of Computer Networks. *Interntional Journal of Computer Applications*, 68(24):17–24, 2013.
- Cisco Press.** Big Data Analytics and NetFlow. <http://www.ciscopress.com/articles/article.asp?p=2437424&seqNum=3>, 2014. [Accessed: 2017-09-29].
- Cisco Systems.** Introduction to Cisco IOS ® NetFlow. http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/prod_white_paper0900aecd80406232.pdf, 2012. [Accessed: 2017-09-25].
- Cisco Systems.** Network as a Security Sensor White Paper - Cisco. <https://www.cisco.com/c/en/us/solutions/collateral/enterprise->

- networks/enterprise-network-security/white-paper-c11-736595.html, 2015a. [Accessed: 2017-09-25].
- Cisco Systems.** The Zettabyte Era: Trends and Analysis. <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html>, 2015b. doi:1465272001812119. [Accessed: 2017-10-15].
- Claise, B., Bryant, S., Leinen, S., Dietz, T., and Trammell, B. H.** RFC5101 Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. <https://tools.ietf.org/html/rfc5101>, 2008. [Accessed: 2017-11-02].
- Dobbins, R.** Mirai IoT Botnet Description and DDoS Attack Mitigation. <https://www.arbornetworks.com/blog/asert/mirai-iot-botnet-description-ddos-attack-mitigation/>, 2016. [Accessed: 2016-12-10].
- Du, Y., Liu, J., Liu, F., and Chen, L.** A real-time anomalies detection system based on streaming technology. In *Proceedings - 2014 6th International Conference on Intelligent Human-Machine Systems and Cybernetics, IHMSC 2014*, volume 2, pages 275–279. 2014. ISBN 9781479949557. doi:10.1109/IHMSC.2014.168.
- Ertoz, L., Eilertson, E., Lazarevic, A., Tan, P.-n., Kumar, V., Srivastava, J., and Dokas, P.** Minds-minnesota intrusion detection system. *Next Generation Data Mining*, pages 199–218, 2004.
- Fachkha, C. and Debbabi, M.** Darknet as a Source of Cyber Intelligence: Survey, Taxonomy, and Characterization. *IEEE Communications Surveys and Tutorials*, vol. 18, page 11971227, 2016.
- FICO.** Understanding All 3 FICO Credit Scores — myFICO. <http://www.myfico.com/credit-education/credit-scores/>, 2017. [Accessed: 2017-09-26].
- François, J., Wang, S., State, R., and Engel, T.** BotTrack: Tracking botnets using netflow and pageRank. *Lecture Notes in Computer Science*, 6640 LNCS(PART 1):1–14, 2011. ISSN 03029743. doi:10.1007/978-3-642-20757-0_1.
- García, S. and Uhler, V.** Malware Capture Facility Projects. <https://mcfp.weebly.com/>, 2017. [Accessed: 2017-11-02].

- Gupta, A., Birkner, R., Canini, M., Feamster, N., Mac-Stoker, C., and Willinger, W.** Network Monitoring as a Streaming Analytics Problem. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks - HotNets '16*, pages 106–112. Atlanta, 2016. ISBN 9781450346610. doi:10.1145/3005745.3005748.
- Herzberg, B., Bekerman, D., and Zeifman, I.** Breaking Down Mirai: An IoT DDoS Botnet Analysis. <https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html>, 2016. [Accessed: 2016-12-10].
- Hofstede, R., Čeleda, P., Trammell, B., Drago, I., Sadre, R., Sperotto, A., and Pras, A.** Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX. *IEEE Communications Surveys and Tutorials*, 16(4):2037–2064, 2014. ISSN 1553877X. doi:10.1109/COMST.2014.2321898.
- Hofstede, R. and Sperotto, A.** SSH Compromise Detection using NetFlow / IPFIX. *ACM SIGCOMM Computer Communication Review*, 44:20–26, 2014. doi:10.1145/2677046.2677050.
- Huston, G.** RFC5398 Autonomous System (AS) Number Reservation for Documentation Use. <https://www.rfc-editor.org/info/rfc5398>, 2008. doi:10.17487/rfc5398. [Accessed: 2017-10-01].
- IBM.** 5 Things to Know About Big Data in Motion (5 Things To Know IBM Redbooks Blog). https://www.ibm.com/developerworks/community/blogs/5things/entry/5_things_to_know_about_big_data_in_motion?lang=en, 2013. [Accessed: 2017-09-25].
- Jirsik, T., Cermak, M., Tovarnak, D., and Celeda, P.** Toward stream-based IP flow analysis. *IEEE Communications Magazine*, 55(7):70–76, 2017. ISSN 01636804. doi:10.1109/MCOM.2017.1600972.
- Kerr, D. R. and Bruins, B. L.** US Patent US Patent US 6243667 B1: Network flow switching and flow data export. 2001.
- Khule, M., Singh, M., and Kulhare, D.** Enhanced Worms Detection By NetFlow. *International Journal Of Engineering And Computer Science*, 3(3):5123–5127, 2014.
- Kovacs, K.** Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Couchbase vs OrientDB vs Aerospike vs Neo4j vs Hypertable vs Elasticsearch vs Accumulo vs VoltDB vs Scalaris comparison. <https://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis>, 2014. [Accessed: 2017-10-01].

- Kreps, J.** Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). <https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>, 2014. [Accessed: 2017-09-27].
- Krmicek, V.** Inspecting DNS Flow Traffic for Purposes of Botnet Detection. *GEANT3 JRA2 T4 Internal Deliverable*, pages 1–9, 2011.
- Lee, J.-H., Kim, I. K., and Han, K.-J.** An Anormal Connection Detection System based on Network Flow Analysis. In *IEEE 5th International Conference on Consumer Electronics*, pages 71–75. Berlin, 2015. ISBN 9781479987481.
- Lee, Y. Y. and Lee, Y. Y.** Toward scalable internet traffic measurement and analysis with Hadoop. *SIGCOMM Computer Communication Review*, 43(1):5–13, 2012. ISSN 01464833. doi:10.1145/2427036.2427038.
- Malaska, T.** Architectural Patterns for Near Real-Time Data Processing with Apache Hadoop. <http://blog.cloudera.com/blog/2015/06/architectural-patterns-for-near-real-time-data-processing-with-apache-hadoop/>, 2015. [Accessed: 2017-11-21].
- Marchal, S., Jiang, X., State, R., and Engel, T.** A big data architecture for large scale security monitoring. In *3rd IEEE International Congress on Big Data, BigData Congress 2014*, pages 56–63. Institute of Electrical and Electronics Engineers Inc., 2014. ISBN 9781479950577 (ISBN). ISSN 2379-7703. doi:10.1109/BigData.Congress.2014.18.
- Marchetti, M., Pierazzi, F., Colajanni, M., and Guido, A.** Analysis of high volumes of network traffic for Advanced Persistent Threat detection. *Computer Networks*, 109:1–15, 2016. ISSN 13891286. doi:10.1016/j.comnet.2016.05.018.
- McAfee, A. and Brynjolfsson, E.** Big Data: The Management Revolution. <https://hbr.org/2012/10/big-data-the-management-revolution>, 2012. [Accessed: 2017-11-01].
- Mikians, J., Dhamdhere, A., Dovrolis, C., Barlet-Ros, P., and Solé-Pareta, J.** Towards a statistical characterization of the interdomain traffic matrix. *Lecture Notes in Computer Science*, 7290 LNCS(PART 2):111–123, 2012. ISSN 03029743. doi:10.1007/978-3-642-30054-7_9.
- Moore, D., Shannon, C., Voelker, G., and Savage, S.** Network telescopes: Technical report. Technical report, CAIDA, 2004. <Http://www.caida.org/publications/papers/2004/tr-2004-04/tr-2004-04.pdf>.

- Murray, D. and Koziniec, T.** The state of enterprise network traffic in 2012. *APCC 2012 - 18th Asia-Pacific Conference on Communications: "Green and Smart Communications for IT Innovation"*, pages 179–184, 2012. doi:10.1109/APCC.2012.6388126.
- Narkhede, N.** Exactly-once Semantics is Possible: Here's How Apache Kafka Does it. <https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>, 2017. [Accessed: 2017-09-26].
- Narsude, C.** Real-Time Event Stream Processing - Paradigms and Low Latency. <https://www.datatorrent.com/blog/real-time-event-stream-processing-what-are-your-choices/>, 2015a. [Accessed: 2017-10-02].
- Narsude, C.** Real-Time Event Stream Processing What Are Your Choices? <https://www.datatorrent.com/blog/real-time-event-stream-processing-what-are-your-choices/>, 2015b. [Accessed: 2017-09-26].
- OpenSoc.** Open Security Operations Center. <http://opensoc.github.io/>, 2014. [Accessed: 2017-11-15].
- Pathirage, M.** Kappa Architecture - Where Every Thing Is A Stream. <http://milinda.pathirage.org/kappa-architecture.com/>, 2017. [Accessed: 2017-09-26].
- Quittek, J., Zseby, T., Claise, B., and Zander, S.** RFC3917 Requirements for IP Flow Information Export (IPFIX). <https://www.rfc-editor.org/info/rfc3917>, 2004. doi:10.17487/rfc3917. [Accessed: 2017-09-25].
- Salazar, P.** OpenSOC: An Open Commitment to Security. <https://blogs.cisco.com/security/opensoc-an-open-commitment-to-security>, 2014. [Accessed: 2017-12-30].
- Schiffman, M.** Correlating NetFlow Data for Proactive Security: Network Notoriety. <http://blogs.cisco.com/security/correlating-netflow-data-for-proactive-security-network-notoriety>, 2012. [Accessed: 2016-12-04].
- Sender Score.** Blacklist Lookup, Email Blacklist Removal Sender Score — Return Path. <https://senderscore.org/rtbl/>, 2017. [Accessed: 2017-09-26].
- Sinha, R., Papadopoulos, C., and Heidemann, J.** Internet Packet Size Distributions : Some Observations. pages 1–7, 2007.

- Sirota, J.** Combat Sophisticated Threats How Big Data and OpenSOC Could Help Big Data Architect/Data Scientist Cisco Security Solutions Practice @JamesSirota.
- Sirota, J. and Dolas, S.** Analyzing 1.2 Million Network Packets per Second in Real-time. <https://www.slideshare.net/Hadoop-Summit/analyzing-12-million-network-packets-per-second-in-realtime>, 2014. [Accessed: 2017-09-29].
- Smeester and Associates.** These Nation-States Are The Top 3 Threat Actors in the Cyber Security Game — Smeester & Associates :: Denver, Colorado USA. <https://smeester.com/2017/02/18/these-nation-states-are-the-top-3-threat-actors-in-the-cyber-security-game/>. [Accessed: 2017-11-17].
- Swarankar, V.** #34 Incorrect calculation of first/last field in master_record.t. <https://sourceforge.net/p/nfdump/bugs/34/>, 2010. [Accessed: 2016-12-01].
- Sweeney, M. and Irwin, B.** A NetFlow Scoring Framework For Incident Detection. In *Telecommunication Networks and Applications Conference (SATNAC) 2017*, pages 300–305. Barcelona, Spain, 2017.
- Trefis.** Why Cisco Is Worth Nearly 15 Times As Much As Juniper. <https://www.forbes.com/sites/greatspeculations/2017/07/03/why-cisco-is-worth-nearly-15-times-as-much-as-juniper/#d2599e2a3fe1>, 2017. [Accessed: 2017-11-17].
- Tzoumas, K.** High-throughput, low-latency, and exactly-once stream processing with Apache Flink. <https://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink>, 2015. [Accessed: 2017-09-26].
- Vaarandi, R. and Pihelgas, M.** Using security logs for collecting and reporting technical security metrics. *Proceedings - IEEE Military Communications Conference MILCOM*, pages 294–299, 2014. doi:10.1109/MILCOM.2014.53.
- Van Der Toorn, O., Hofstede, R., Jonker, M., and Sperotto, A.** A first look at HTTP(S) intrusion detection using NetFlow/IPFIX. *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015*, pages 862–865, 2015. doi:10.1109/INM.2015.7140395.
- Walker, M.** Batch vs. Real Time Data Processing - Data Science Central. <http://www.datascienceassn.org/content/batch-vs-real-time-data-processing>, 2015. [Accessed: 2017-11-21].

- Wang, L. and Jones, R.** Big Data Analytics for Network Intrusion Detection: A Survey. *International Journal of Networks and Communications*, 7(1):24–31, 2017. doi:10.5923/j.ijnc.20170701.03.
- Wilson, C.** The Difference Between Real Time, Near-Real Time, and Batch Processing in Big Data. <http://blog.syncsort.com/2015/11/big-data/the-difference-between-real-time-near-real-time-and-batch-processing-in-big-data/>, 2015. [Accessed: 2017-11-21].
- Zhao, S., Chandrashekar, M., Lee, Y., and Medhi, D.** Real-time network anomaly detection system using machine learning. *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*, (October 2016):267–270, 2015. doi:10.1109/DRCN.2015.7149025.
- Zuech, R., Khoshgoftaar, T. M., and Wald, R.** Intrusion detection and Big Heterogeneous Data: a Survey. *Journal of Big Data*, 2(1), 2015. ISSN 21961115. doi: 10.1186/s40537-015-0013-4.

Appendix A - Flow DDL

The following listing is the SQL data definition language for the NetFlow data staging table. This illustrates the basic fields used from the NetFlow records by the Themis framework.

```
1 CREATE TABLE flow_data (  
2   id bigint(20) NOT NULL AUTO.INCREMENT,  
3   timestamp datetime DEFAULT NULL,  
4   protocol int(11) DEFAULT NULL,  
5   src_ip char(15) DEFAULT NULL,  
6   dst_ip char(15) DEFAULT NULL,  
7   src_port int(11) DEFAULT NULL,  
8   dst_port int(11) DEFAULT NULL,  
9   packets bigint(20) DEFAULT NULL,  
10  bytes bigint(20) DEFAULT NULL,  
11  flags char(6) DEFAULT NULL,  
12  PRIMARY KEY (id)  
13 )
```

Listing A.1: Flow DDL

Appendix B - Input JSON Schema

The listing below contains the JSON schema definition for input records for the framework. All flow data ingested by the Kafka queue for processing in the Storm topology must adhere to this schema.

```
1 {
2   "$schema": "http://json-schema.org/draft-04/schema#",
3   "definitions": {},
4   "properties": {
5     "bps": {
6       "type": "integer"
7     },
8     "bytes": {
9       "type": "integer"
10    },
11    "dst_ip": {
12      "type": "string"
13    },
14    "dst_port": {
15      "type": "integer"
16    },
17    "duration_unix_secs": {
18      "type": "integer"
19    },
20    "end_timestamp_unix_secs": {
21      "type": "integer"
22    },
23    "flags": {
24      "type": "string"
25    },
26    "id": {
27      "type": "integer"
28    },
29    "packets": {
30      "type": "integer"
31    },
32    "pps": {
33      "type": "integer"
34    },
35    "protocol": {
36      "type": "integer"
37    },
38    "src_ip": {
39      "type": "string"
40    },
41    "src_port": {
42      "type": "integer"
43    },
44    "start_timestamp_unix_secs": {
45      "type": "integer"
46    },
47    "tos": {
48      "type": "integer"
49    }
50  },
```

```
51 "type": "object"  
52 }
```

Listing B.1: Flow JSON

The following listing illustrates an example of a flow data record that conforms to the input schema.

```
1 {  
2   duration_unix_secs    : 0,  
3   src_ip    : 192.0.2.198 ,  
4   dst_ip    : 187.121.128.84 ,  
5   src_port  : 23,  
6   dst_port  : 37906,  
7   protocol  : 6,  
8   bytes    : 340,  
9   packets  : 6,  
10  tos      : 0,  
11  flows    : 0,  
12  pps      : 0,  
13  bps      : 0,  
14  flags    : .A.RS. ,  
15  start_timestamp_unix_secs : 1484033957,  
16  end_timestamp_unix_secs   : 1484033957,  
17  id       : 20  
18 }
```

Listing B.2: Flow JSON

Appendix C - Java Flow Tuple Objects

The following snippet of code from the Java source code illustrates the attributes stored in the Storm tuples. The ingested JSON flow records are converted to a corresponding Java object and the extra fields are populated as the tuple is processed in the Storm topology.

```
1 public class NetFlow implements Serializable {
2     private static final long serialVersionUID = 1L;
3
4     // control attributes
5     long id;
6     long start_processing_ms;
7     int splitter_instances = 0;
8     int cache_counter = 0;
9
10    // NetFlow attributes
11    long start_timestamp_unix_secs;
12    long end_timestamp_unix_secs;
13    long duration_unix_secs;
14    short protocol;
15    String src_ip;
16    String dst_ip;
17    long src_ip_int;
18    long dst_ip_int;
19    int src_port;
20    int dst_port;
21    long packets_sent;
22    long bytes_sent;
23    long packets_recv;
24    long bytes_recv;
25    int flows;
26    String flags;
27    short tos;
28    int bps;
29    int pps;
30
31    // geoip data
32    String src_city;
33    String src_country;
34    String src_country_code;
35    String dst_city;
36    String dst_country;
37    String dst_country_code;
38    Double src_latitude;
39    Double src_longitude;
40    Double dst_latitude;
41    Double dst_longitude;
42
43    // ISP data
44    int src_as;
45    int dst_as;
46    String flow_direction;
47
48    // score related attributes
49    int goodness;
50    int badness;
```

```
51     Set<FlowScore> good_scores;  
52     Set<FlowScore> bad_scores;  
53  
54     // setters and getters left out for brevity  
55 }
```

Listing C.1: Flow Tuple

As a flow record is scored a FlowScore object is added to the Set of FlowScores in the corresponding tuple record. The FlowScore Java class attributes are illustrated in the following listing.

```
1 public class FlowScore {  
2     private static final long serialVersionUID = 1L;  
3  
4     String    score_category;  
5     String    score_code;  
6     int    score;  
7  
8     // setters and getters left out for brevity  
9 }
```

Listing C.2: Flow Score Class

Appendix D - Output JSON Schemas

Once processing is completed the Java tuple is converted back to JSON and emitted from the Storm topology. The following listing contains the JSON schema for emitted scored flows.

```
1 {
2   "$schema": "http://json-schema.org/draft-04/schema#",
3   "definitions": {},
4   "id": "http://example.com/example.json",
5   "properties": {
6     "badness": {
7       "id": "/properties/badness",
8       "type": "integer"
9     },
10    "bps": {
11      "id": "/properties/bps",
12      "type": "integer"
13    },
14    "bytes_recv": {
15      "id": "/properties/bytes_recv",
16      "type": "integer"
17    },
18    "bytes_sent": {
19      "id": "/properties/bytes_sent",
20      "type": "integer"
21    },
22    "dst_as": {
23      "id": "/properties/dst_as",
24      "type": "integer"
25    },
26    "dst_ip": {
27      "id": "/properties/dst_ip",
28      "type": "string"
29    },
30    "dst_port": {
31      "id": "/properties/dst_port",
32      "type": "integer"
33    },
34    "duration_unix_secs": {
35      "id": "/properties/duration_unix_secs",
36      "type": "integer"
37    },
38    "end_timestamp_unix_secs": {
39      "id": "/properties/end_timestamp_unix_secs",
40      "type": "integer"
41    },
42    "event_time_dt": {
43      "id": "/properties/event_time_dt",
44      "type": "null"
45    },
46    "event_time_str": {
47      "id": "/properties/event_time_str",
48      "type": "string"
49    },
50    "flags": {
```

```

51     "id": "/properties/flags",
52     "type": "string"
53 },
54 "flow_direction": {
55     "id": "/properties/flow_direction",
56     "type": "string"
57 },
58 "flow_id": {
59     "id": "/properties/flow_id",
60     "type": "integer"
61 },
62 "flows": {
63     "id": "/properties/flows",
64     "type": "integer"
65 },
66 "goodness": {
67     "id": "/properties/goodness",
68     "type": "integer"
69 },
70 "packets_recv": {
71     "id": "/properties/packets_recv",
72     "type": "integer"
73 },
74 "packets_sent": {
75     "id": "/properties/packets_sent",
76     "type": "integer"
77 },
78 "pps": {
79     "id": "/properties/pps",
80     "type": "integer"
81 },
82 "protocol": {
83     "id": "/properties/protocol",
84     "type": "integer"
85 },
86 "src_as": {
87     "id": "/properties/src_as",
88     "type": "integer"
89 },
90 "src_ip": {
91     "id": "/properties/src_ip",
92     "type": "string"
93 },
94 "src_port": {
95     "id": "/properties/src_port",
96     "type": "integer"
97 },
98 "start_timestamp_unix_secs": {
99     "id": "/properties/start_timestamp_unix_secs",
100     "type": "integer"
101 },
102 "timestamp": {
103     "id": "/properties/timestamp",
104     "type": "integer"
105 },
106 "tos": {
107     "id": "/properties/tos",
108     "type": "integer"
109 },
110 "type": {
111     "id": "/properties/type",
112     "type": "string"
113 }
114 },
115 "type": "object"
116 }

```

Listing D.1: Scored Flow JSON

For each score and tag assigned to a flow a separate record is emitted. The following listing contains this JSON schema.

```

1 {
2   "$schema": "http://json-schema.org/draft-04/schema#",
3   "definitions": {},
4   "id": "http://example.com/example.json",
5   "properties": {
6     "bad_score": {
7       "id": "/properties/bad_score",
8       "type": "integer"
9     },
10    "bad_score_tag": {
11      "id": "/properties/bad_score_tag",
12      "type": "string"
13    },
14    "bps": {
15      "id": "/properties/bps",
16      "type": "integer"
17    },
18    "bytes_recv": {
19      "id": "/properties/bytes_recv",
20      "type": "integer"
21    },
22    "bytes_sent": {
23      "id": "/properties/bytes_sent",
24      "type": "integer"
25    },
26    "dst_as": {
27      "id": "/properties/dst_as",
28      "type": "integer"
29    },
30    "dst_ip": {
31      "id": "/properties/dst_ip",
32      "type": "string"
33    },
34    "dst_port": {
35      "id": "/properties/dst_port",
36      "type": "integer"
37    },
38    "flags": {
39      "id": "/properties/flags",
40      "type": "string"
41    },
42    "flow_direction": {
43      "id": "/properties/flow_direction",
44      "type": "string"
45    },
46    "flow_id": {
47      "id": "/properties/flow_id",
48      "type": "integer"
49    },
50    "flows": {
51      "id": "/properties/flows",
52      "type": "integer"
53    },
54    "good_score": {
55      "id": "/properties/good_score",
56      "type": "integer"
57    },
58    "good_score_tag": {
59      "id": "/properties/good_score_tag",
60      "type": "null"
61    },
62    "packets_recv": {
63      "id": "/properties/packets_recv",
64      "type": "integer"
65    },
66    "packets_sent": {
67      "id": "/properties/packets_sent",
68      "type": "integer"
69    },
70    "pps": {
71      "id": "/properties/pps",
72      "type": "integer"
73    },
74    "protocol": {
75      "id": "/properties/protocol",

```

```
76     "type": "integer"
77   },
78   "src_as": {
79     "id": "/properties/src_as",
80     "type": "integer"
81   },
82   "src_ip": {
83     "id": "/properties/src_ip",
84     "type": "string"
85   },
86   "src_port": {
87     "id": "/properties/src_port",
88     "type": "integer"
89   },
90   "start_timestamp_unix_secs": {
91     "id": "/properties/start_timestamp_unix_secs",
92     "type": "integer"
93   },
94   "timestamp": {
95     "id": "/properties/timestamp",
96     "type": "integer"
97   },
98   "tos": {
99     "id": "/properties/tos",
100    "type": "integer"
101  },
102  "type": {
103    "id": "/properties/type",
104    "type": "string"
105  }
106 },
107 "type": "object"
108 }
```

Listing D.2: Scored Flow Detail JSON

Appendix E - Logger Bolt Output

The following listing contains an example of the logging bolt output.

```
1 2017-08-28 09:13:26.527 c.v.n.b.o.LoggerBolt Thread-74-print-messages-tcp-executor[137 137] [INFO]
  LoggerBolt task id 137 / default - record id = 1819 [2397]
  {-6683385491907144903=-5411684369963292021} / {"id":1819,"start_processing_ms":1503904404130,"
  splitter_instances":3,"cache_counter":3,"start_timestamp_unix_secs":1484038190,"
  end_timestamp_unix_secs":1484038190,"duration_unix_secs":0,"protocol":6,"src_ip":"31.13.70.52","
  dst_ip":"192.0.2.130","src_ip_int":520963636,"dst_ip_int":3289772674,"src_port":443,"dst_port"
  :59183,"packets_sent":25,"bytes_sent":18917,"packets_rcv":27,"bytes_rcv":3098,"flows":0,"flags":
  ".AP.SF","tos":0,"bps":0,"pps":0,"src_city":"Los Angeles","src_country":"United States","
  src_country_code":"US","dst_city":"Grahamstown","dst_country":"South Africa","dst_country_code":"
  ZA","src_latitude":34.0544,"src_longitude":-118.244,"dst_latitude":-33.3,"dst_longitude":26.5333,"
  src_as":32934,"dst_as":2018,"flow_direction":"INBOUND","goodness":30,"badness":10,"good_scores":[{"
  "score_category":"IP-LIST","score_code":"NDPLGOOD","score":30}], "bad_scores":[{"score_category":"
  SUSPICIOUS-TRAFFIC","score_code":"DARK_IP","score":10}]}
```

Listing E.1: Log Output

Appendix F - Bolts, Tags and Scores

The table in this chapter contains a list of all configured scoring bolts, the associated scores and the tags. Negative values for scores indicate a bad score and positive values a good score. If a bolt is listed with both a good and a bad score then this indicated negative scoring takes place in the bolt instance.

Table F.1: Scoring bolts detail

Scoring Bolts Tags and Scores			
Bolt Instance	Bolt	Tag	Score
score-dark-ip	ScoreDarkIPBolt	SUSPICIOUS_TRAFFIC/DARK_IP	-40
score-emerging-threats-list	ScoreGenericIPListBolt	IP_LIST/ EMERGING.THREATS	-50
score-alienvault-list	ScoreGenericIPListBolt	IP_LIST/ALIENVAULT	-50
score-suspect-country	ScoreCountryBolt	COUNTRY/SUSPECT	-80
score-intelmq	ScoreIntelMQBolt	INTELMQ/INTELMQ	-70
score-ndpi-known-list	ScoreGenericIPListBolt	IP_LIST/NDPI.GOOD	100
score-syn-only	ScorePossibleScanBolt	POSSIBLE_SCAN/SYN_ONLY	-60
score-ssh-brute	ScoreSSHBruteforceBolt	BRUTE_FORCE/SSH	90
score-http-brute	ScoreHTTPBruteforceBolt	BRUTE_FORCE/HTTP	90
score-tcp-hosted-service	ScoreServiceBolt	HOSTED_SERVICES/TCP	+100/-50
score-tcp-remote-service	ScoreServiceBolt	REMOTE_SERVICES/TCP	+100
score-udp-hosted-service	ScoreServiceBolt	HOSTED_SERVICES/UDP	+100/-50
score-udp-remote-service	ScoreServiceBolt	REMOTE_SERVICES/UDP	+100
score-bad-udp-ports	ScoreInsecurePortConversationBolt	PORT_LIST/INSECURE_TCP_TRAFFIC	-50
score-bad-tcp-ports	ScoreInsecurePortConversationBolt	PORT_LIST/UNKNOWN_TCP_TRAFFIC	-70
score-bad-udp-traffic	ScoreUnknownPortConversationBolt	PORT_LIST/INSECURE_UDP_TRAFFIC	-50
score-bad-tcp-traffic	ScoreUnknownPortConversationBolt	PORT_LIST/UNKNOWN_UDP_TRAFFIC	-70

Appendix G - Scoring Topology

The following figure illustrates the complete topology implemented for this work. The blue coloured bolt is the Kafka ingestion bolt, the yellow coloured bolts are utility bolts, red and green coloured bolts are scoring bolts and the orange bolts are output bolts.

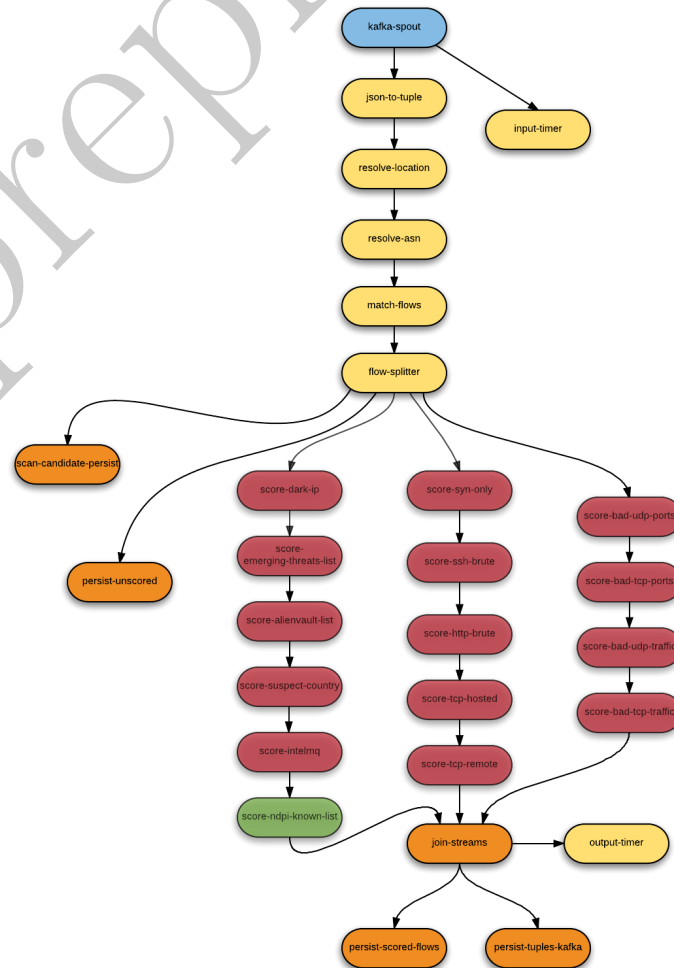


Figure G.1: Flow scoring topology

Appendix H - Source Code

The source code for this project can be obtained from <https://github.com/sweeneymj/themis>.