석사 학위논문
Master's Thesis

랜덤 포레스트를 이용한 하둡기반 P2P 봇넷 트래픽 분류 기법

Hadoop-based P2P Botnet Traffic Classification Using Random Forests

Khalid Huseynov (칼리드 후세이너브)

전산 학과
Department of Computer Science

KAIST

2014

랜덤 포레스트를 이용한 하둡기반 P2P 봇넷 트래픽 분류 기법

Hadoop-based P2P Botnet Traffic Classification Using Random Forests

Hadoop-based P2P Botnet Traffic Classification Using Random Forests

Advisor : Professor Kim, Kwang Jo

by

Huseynov, Khalid
Department of Computer Science
Korea Advanced Institute of Science and Technology (KAIST)

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science. The study was conducted in accordance with Code of Research Ethics[1]

2014. 12. 19

Approved by

Professor Kim, Kwang Jo (_______)

[Advisor]

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

Hadoop-based P2P Botnet Traffic Classification Using Random Forests

Huseynov, Khalid

위 논문은 한국과학기술원 석사 학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2014 년 12 월 19 일

심사위원장	Kwangjo Kim (인)
심사위원	Young Hee Lee (인)
심사위원	Soontae Kim (인)

MCS
20134361

Khalid Huseynov. 칼리드 후세이너브. Hadoop-based P2P Botnet Traffic Classification Using Random Forests. 랜덤 포레스트를 이용한 하둡기반 P2P 봇넷 트래픽 분류 기법. Department of Computer Science. 2014. 33 p. Advisor Prof. Kwangjo Kim.

ABSTRACT

During the last decade a number of coordinated security breaches happened on a global scale. Botnets represent major infrastructure for such coordinated cyber-attacks on citizens, enterprises, and governments. Botnet can be referred as a large network of compromised computers being remotely controlled. Many recent countermeasures utilize machine-learning techniques due to its adaptability and “model-free” properties. Due to high volumes of traffic, the challenge is posed by managing tradeoff between system scalability and accuracy. We propose a novel Hadoop-based P2P botnet detection and classification method solving the problem of scalability and having high accuracy. Note that proposed system can classify not only P2P botnet traffic but also traffic generated by legal P2P applications (e.g. Skype, eMule). Our system bridges the gap between state-of-the-art P2P traffic detection/classification methods and the corresponding research in distributed Big Data processing. Hadoop was chosen as main development framework for the whole system. Moreover, random forests ensemble method was employed for the categorization of the traffic. Inherent distributed characteristics of the random forests classifier add more scalability to the system.

Keywords: Botnets, Hadoop, Traffic Classification, P2P.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 What is Botnet?

In the age of growing communication networks, the attack and corresponding defense mechanisms become ingenious to a greater extent. A decade ago, attacks were mainly correlated with skilled individuals aiming for recognition, whereas, currently, a well-established underground economy nourishes most of large-scale malicious activities [1, 15]. Majority of those activities are carried out by the means of botnets. For example, spam activities have been estimated to cause financial loss of US$ 100 billion to global economy in 2007 [3].

A botnet represents a group of compromised host machines called bots which are controlled remotely by a botmaster server [2]. The botmaster is not only able to download any confidential files from the infected host but also capable to execute any malicious code on the infected machines, which turns the botnet into a platform of massively coordinated cyber-attacks. Bots can perform any kinds of malicious attacks such as Distributed Denial of Service (DDoS), click-fraud, adware, spreading spam, key logging, and stealing personal information. According to report by Damballa, a cyber-security company, few millions of computers in the United States were infected by botnets in 2009 (e.g., 3.6M by Zeus botnet) [4].

The origin of botnet like systems emerged in 1993 and was based on a text chatting system organized in communication channels, namely Internet Relay Chat (IRC). This early IRC bot was named Eggdrop [5] and further developed into a platform for distributed denial of service (DDoS) attacks.

1.2 Architecture and Communication Protocols

The two key components of botnets include the protocols employed for communication between bots and botmaster and the architecture of bot network. The botnets in the early 2000's [5, 6] utilized IRC as a communication protocol, which refers to centralized architecture. In this scenario, botmaster is able to communicate with its bots in real time via chat, through IRC-based Command and Control (C&C) server. In the middle of 2000's, HTTP-based centralized botnet architectures [7, 8] emerged. In such architectures, bots periodically contact their C&C server to receive further instructions using HTTP protocol as a basis. Figure 1 represents general architecture of such a centralized bot network. The main drawback of centralized architecture is the ease of take down by mere shutdown of C&C server.

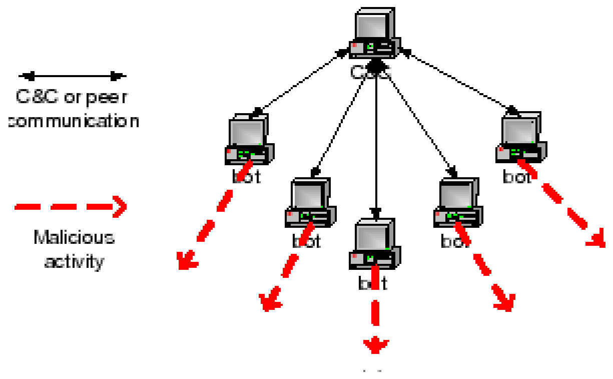


Figure 1. Centralized botnet architecture.

Alternatively, botnets also utilize Peer to Peer (P2P) architecture and protocols that evolved in the mid-2000s. In this scheme, the commands of botmaster can pass through multiple bots in order to reach their destination, and if some of the bots are down, another path is selected based on its P2P protocol. The routing in this P2P topology is enabled by the means of distributed hash tables (DHTs) without any need for DNS address resolution. Kademlia [9] is a good example of such a DHT algorithm, and it is utilized in most P2P applications such as eMule, BitTorrent, and Overnet. Nugache, Storm and Waledac [10, 11] are also well-known

P2P botnets utilizing DHTs for P2P routing. Figure 2 represents an example of P2P botnet architecture.

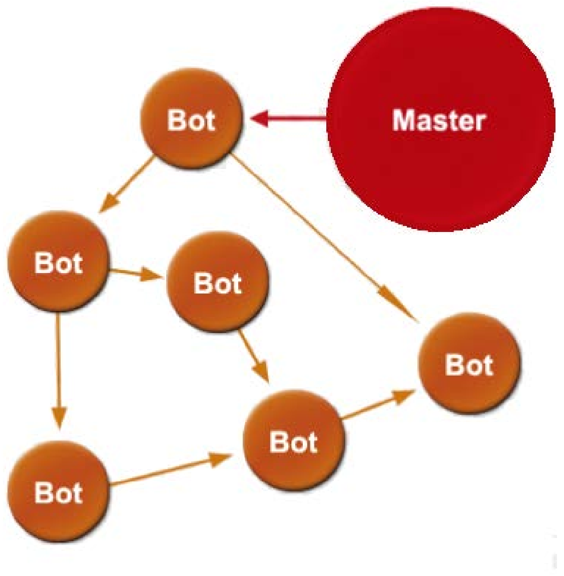


Figure 2. P2P botnet architecture.

More advanced hybrid architectures employ advantages of centralized as well distributed schemes. An example of such advanced architecture was a topic of research in [12]. It was found that hybrid topologies have two types of bots: servant and client bots. Servant bots behave as servers and clients concurrently, whereas client bots are usually located behind the firewalls and connect to their local servants periodically to return the new commands from the botmaster.

1.3 Botnet Lifecycle.

Predominantly, botnet lifecycle is categorized into five major phases: initial infection, secondary infection, connection to C&C, malicious activities, and maintenance phases. These phases are sometimes named differently or merged into fewer phases, depending on authors' preferences [13, 14]. Figure 3 represents overall diagram of botnet lifecycle.

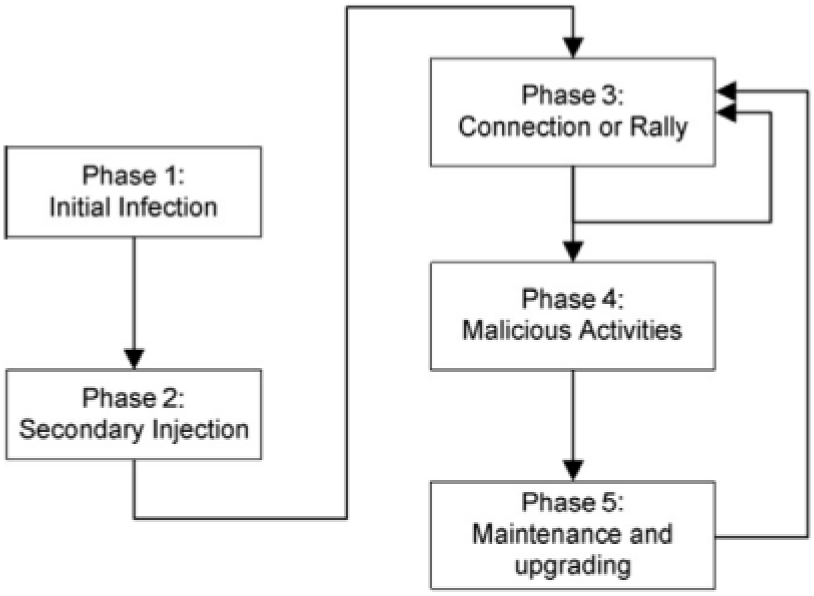


Figure 3. Botnet lifecycle.

The Initial Infection phase is realized in the same way as many hosts get infected, namely system vulnerabilities, email attachments, drives-by-download, infected memory carriers, etc. [10, 13, 14]. This phase is crucial for the progress of the whole infection; if failed, further infection is impossible.

After successful Initial Infection, the hijacked host contacts the malware server to download the real bot executable. This phase is named as Secondary Injection since the binary having all bot logic gets executed on this stage. Binary download may be realized by the means of FTP, HTTP or P2P protocols [10, 13, 14].

Further, infected host (bot) contacts with C&C server in order to register as online bot and retrieve the commands to be executed. Bot behavior from this point depends on the retrieved commands from C&C server [16]. By following commands, bot can further transit into Malicious Activities phase. As previously mentioned, those activities may include DDoS attacks, click-fraud, id theft, spreading spam, key logging, etc. Once malicious activity executed, bot may contact C&C server again for new set of commands. Also, bot may contact C&C server periodically while performing previously assigned malicious activities.

Lastly, as any software, bot code is maintained as well as updated [13, 14, 16]. This can be due to adding new features, fixing bugs, moving to new C&C server, etc. Bot maybe vulnerable at this stage since it'll have similar network behavior for all hosts connected to botnet. Once updated, bot will transit into Connection phase.

1.4 Introduction to Distributed Computing: MapReduce Paradigm.

MapReduce [17] is a distributed computing framework developed at Google for processing large unstructured datasets. The idea is to split large dataset into chunks and process each chunk in parallel by followed merge process. The framework alleviates programmer's job of managing inter-process communication, data distribution, load balancing and machine failures. Thus, the code written for processing a megabyte of data can be easily extended for terabytes or even petabytes of data. For example, Google's clusters process more than twenty petabytes of data logs on a daily basis [17].

MapReduce programming model was inspired from *map* and *reduce* functions extensively employed in Lisp-like functional languages. Thus, each MapReduce job has two main methods, namely Map and Reduce, that can be implemented by programmers. The main objective of Map phase is to turn input records into key/value pairs. Further, all pairs having same key are directed into the same reduce function. The logical view of this process can be represented as follows.

Map (k1, v1) → list (k2, v2)

Every input record is represented by a pair of k1/v1, and Map function is applied to every input record. In Map phase, normally all the data is in v1, and k1 is used as a reference to record. Once a pair of k1/v1 is processed by Map function, the output pair of k2/v2 is produced. Further, all the values with the same k2 are collected together and directed into Reduce phase as shown below.

Reduce (k2, list (v2)) $\rightarrow$ list (v3)

The Reduce method accepts the key (k2) and the list of values (list(v2)) associated with k2. This list is processed in Reduce method according to the logic of program. Lastly, a list of final values (list(v3)) is returned. The following Figure 4 shows a simple application of MapReduce paradigm in the context of WordCount problem [19].

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    output (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += ParseInt(pc)
  output (word, sum)
```

Figure 4. Implementation of WordCount problem in MapReduce.

Given this example, k1 corresponds to the document name, and v1 corresponds to the document itself. The logic of Map function parses document and extracts all the words from it. For every word in document, Map function outputs word as a key and value of “1”, meaning this word occurred once at a given time in document.

Further, Reduce function accepts the word as a key, and a list of “1”s as value. Then, those “1”s are summed up and the final occurrence count of word is given. Note that in the given example we have “partialCounts” as a value. The reason is that MapReduce has also optional intermediary Combine phase before applying Reduce. This Combine phase applies the Reduce function on the local machines before globally transferring the results into Reduce phase. This is done in order to minimize inter-process communication as much as possible.

For our purposes, we used an open-source implementation of MapReduce, namely Hadoop [18], which is maintained by the Apache Software Foundation.

Chapter 2. Related Work

2.1 Overview of Botnet Detection Techniques

Approaches in botnet detection can be categorized into one of the two main methods: signature-based or anomaly-based technique [13, 16]. Signature-based systems utilize certain patterns pertinent to specific botnets. For example, a specific byte signature in packet header is one of the common ways employed in signature-based detection systems. The advantages of signature-based detection can be referred as ease of use once signature is ready and speed of detection. However, numerous disadvantages can be listed as well. First of all, this method isn’t flexible, and even minor changes in signature may undermine the whole detection process. This means that *zero-day* attacks are unavoidable. Moreover, having large database of signatures may have opposite effect of system slowdown. Secondly, the devising signature for a new botnet is always a cumbersome job without guarantee of having any signature at all. Lastly, signature-based detection is almost impossible once the botnet traffic is encrypted.

On the other hand, anomaly-based approaches have more flexibility and are able to detect different types of botnets by following similar set of rules. Further, this set of approaches has better performance in the context of novelty detection. Perhaps this is the reason why most of the recent approaches in botnet detection can be categorized as anomaly-based approaches [13, 16]. Figure 5 shows general categorization of botnet detection techniques. Anomaly-based detection techniques can be further classified into host- and network-based techniques.

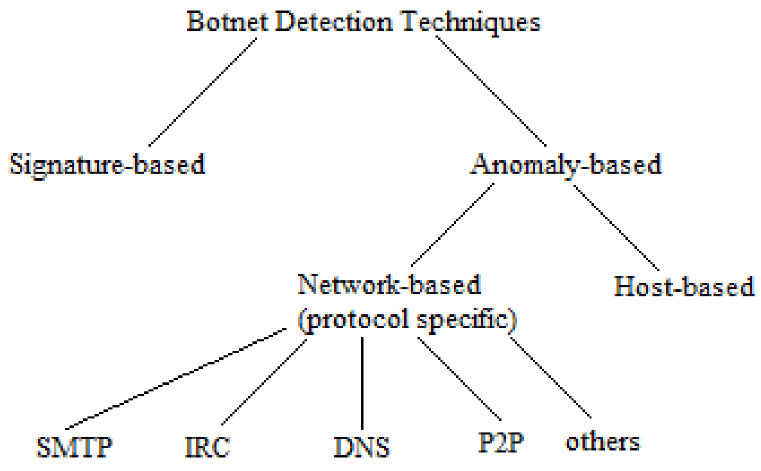

Botnet Detection Techniques
Signature-based
Anomaly-based
Network-based (protocol specific)
Host-based
SMTP
IRC
DNS
P2P
others

Figure 5. General categorization of botnet detection techniques.

Normally host-based detection techniques should be installed on every host. Main principle of host-based botnet detection system is monitoring of the system behavior (*e.g.* system calls, file system, local network behavior) with the purpose of differentiating botnet process from legitimate one [20, 21]. One of the first influential works in this field has been done by Forrest et al. [20] in 1996. They defined a “sense of self” for Unix processes in terms of system call behavior. Then any processes not following the normal behavior could be flagged as malicious one. In this way, they were able to detect several common intrusion including *sendmail* and *lpr*. More recent work by Lanzi et al. [21] utilized similar anomaly-based approach with larger test data. They have proven effectiveness of this approach with almost no false positives and ability to detect most of malware samples. General problem with host-based detection methods is the overhead of software installation on every host, whereas network-based approaches require installation only on network entry points. Further we'll discuss more on network-based botnet detection techniques.

2.1.1 Signature-based Detection Techniques

Signature-based detection techniques are most widely used in real industrial systems [22]. The main reason behind this wide acceptance is high accuracy for the detection of known attacks, although new types of attacks remain undetected. On the other hand, if anomaly-based detection is employed, high occurrence of false positives makes system administrators to manually evaluate all falsely detected breaches [22]. This adds high cost to the usage of anomaly-based detection systems, leaving a room to utilization of signature-based detection systems.

Snort [23] is one of the first signature-based state-of-the-art Network Intrusion Detection Systems (NIDSs), which is still widely utilized nowadays. Snort is lightweight and can easily be configured according to the network administrator's needs. At the heart of the Snort lies its Detection Engine [23] that utilizes special detection rules. Note that it's possible to add Snort rules with the purpose of botnet detection as well. Thus, Snort is a general detection system that can be configured according to the detection rules.

Bro [24] is another signature-based NIDS developed approximately at the same time with Snort. For comparison with Snort, Bro can handle higher speeds and detection rules for Bro can be more complicated [25]. Moreover, Bro is more flexible with its own Bro network programming language, which supports writing any detection rules in form of scripts. Therefore, botnet detection rules can be easily embedded into Bro IDS.

One of the scalable approaches for signature-based IDS was presented in Kargus [26]. This approach accelerates the most computation intensive part of IDS, namely string matching process, in GPU. Snort was used as the basis for the detection engine. A speed close to 40Gbit/sec was obtained during evaluation of this approach. Note that Kargus can be used as signature-based botnet detection system once the botnet detection rules are formalized.

There are pure signature-based botnet detection systems as well. Rishi [27] is a well-known signature-based botnet detector in IRC channels. It has been built on the concept that host machines after being infected contact their C&C server with their nicknames. Here, nicknames are used for further identification of the bots in the botnet. Such nicknames usually contain some constant parts, which are the same for all the bots in the same botnet. Such simple idea was shown to be effective for IRC botnet detection in a network with its speed up to 10Gbit/sec.

A successful attempt to infiltrate a huge botnet system called Torpig was conducted in [28]. This work is of particular interest since they infiltrated Torpig C&C server and were able to record all the communications of its bots. They successfully identified 1.2 million IP addresses of bots, which are connected to an infiltrated C&C server. This proactive approach is an infiltration method rather than reactive signature-based detection

2.1.2 Network Anomaly-based Detection Techniques

There is a multitude of anomaly-based approaches to botnet detection and overall to intrusion detection systems compared to signature-based techniques. Those approaches employ statistical inferences, data mining and machine learning algorithms, graph theory, correlation techniques, information theory, etc. [13].

BotHunter [29] is one of the methods based on the analysis of network traffic, and it relies on botnet lifecycle activities, namely scanning, infection, binary download, and C&C scanning. BotHunter utilizes a Snort-based intrusion detection system for detecting any kind of scanning activities. Once successfully detected, it inspects the payload of flow for other malicious activities from botnet lifecycle. Encrypted packages are the main obstacles for effective functioning of this system.

Further, Bothunter was enhanced into BotMiner [30], which is built on the assumption that all the bots in the botnet exhibit similar network behavior. BotMiner searches and clusters similar connections together using a C-plane monitor. Similar activities are clustered using an A-plane monitor. Then, BotMiner cross-correlates the two planes and finds the C-plane clusters that behave maliciously. Note that traditional clustering methods were used in BotMiner, and results showed the accuracy of 99% with false positive rate below 1%.

Wang et al. [31] proposed another traffic analysis model for P2P botnet detection. They observed that botnet control flows are relatively more stable compared to normal flows. As a result, their algorithm was able to detect the bots using encrypted communication with high true positive rate and low false alarms.

A novel approach based on both traffic behavior analysis and flow intervals was proposed in [32]. The flows were organized based on time intervals. In addition, a machine-learning based classifier was built based on the extracted traffic flow features. This approach was evaluated on a labeled dataset containing malicious traffic from Storm and Waledac P2P botnets, and showed 99% accuracy and less than 1% of false alarm rate.

ProVeX [33] is yet another botnet detection system that is able to work with encrypted traffic. Since most of the encryption schemes utilized by botmasters are simple *XOR*-based symmetric ciphers [33], the authors were able to decrypt the botnet messages with keys obtained by reverse engineering malware code. Second challenge was to differentiate between normal and malicious traffic. Authors applied *probabilistic vectorized signatures* learned from known botnet communications. This is effective method since botnet protocols normally made up either by *positional fields* or *tagged fields*. Tagged fields can be easily recognized by signature-based systems (e.g. "User-Agent: Mozilla"). On the other hand, the probabilistic signatures can detect the communication protocols employing the positions of bytes for representing meaning for them. As a result, this approach showed high accuracy with the speed up to multiple Gbit/s.

A scalable system for stealthy P2P botnet detection was introduced in [34]. This approach is practical due to usage of streaming clustering algorithm (BIRCH [35]) in the process of detection. In the first stage, P2P hosts that are likely to participate in P2P communication are detected. In the second stage, statistical fingerprints are derived based on the differences among communication patterns of various P2P applications. This approach reaches a balance between scalability, speed, and accuracy by utilizing unsupervised and lightweight computational schemes.

More practical perspective to network traffic analysis was shown in PeerRush [36] system. PeerRush is able not only to detect P2P botnets but also classify any kind of P2P traffic running in the network. This is more practical since many network administrators would like to know the application layer software (possibly malicious) running on the controlled subnet. They were able to accurately classify five types of legal P2P applications (e.g. eMule, Bittorent) and three types of P2P botnets (e.g. Storm, Waledac). Moreover, the classification system of the PeerRush is modular and any new threat models can be added without reconfiguration of the system. Our research was partially inspired by PeerRush since network traffic profiling is an important issue for the industrial systems. Moreover, the profiling system scalable under huge volumes of traffic is even more necessary.

2.2 Hadoop-based Botnet Detection Techniques

With the advent of open source distributed computing frameworks such as Hadoop, many systems have been re-implemented on this framework in order to reap the advantages of large scale computing. Thus, the scalability, fault tolerance, and load balancing issues of the detection systems could now be delegated to the underlying framework. This gives a separation of concerns and perspective for innovation in both directions.

BotGraph [37] is one of the first systems having utilized the MapReduce paradigm in spamming bot-users detection. The main observation behind the study is that different bot-users while sending spam from the infected bot computer share same IP address. BotGraph detects this abnormal overlap in IP addresses of large number of users. The bot-users are modeled as nodes of the large graph, and the problem is reduced to finding tightly connected sub graph components in the constructed large user-user graph.

BotCloud [38] is another detection method utilizing large graph processing abilities of Hadoop. They have adopted the PageRank algorithm in the context of P2P botnet detection, meaning nodes having high page rank are possible P2P hosts. Further they differentiate between legal P2P hosts and botnet infected ones by assigning more weight to interconnectedness of the nodes. Thus, highly interconnected nodes forming partition are more probable to be part of botnet.

Big Data analytics framework for P2P botnet detection was built in [39] on top of Hadoop framework. Tshark [40] was used for extracting basic fields from the raw “pcap” files into structured table. Further, features were extracted from the table using Apache Hive [41], which is a query language similar to SQL built on top of Hadoop. Lastly, random forests classifier was built in Apache Mahout [42], which is a machine learning framework on top of Hadoop, for detecting botnet traffic. The results showed accuracy of 99.8% with near real time traffic processing.

Furthermore, Hadoop-based traffic processing and analysis tools have been recently developed [43, 44]. Originally Hadoop did not support parallel processing of *pcap* files directly from HDFS. This is the reason why earlier research in [39] employed Tshark for parsing the pcap files. However authors in [43] were able to develop binary input format, namely PcapInputFormat, for processing pcap files from HDFS on Hadoop. Moreover, they showed higher performance compared to the state-of-the-art traffic processing tools [45, 46]. We used PcapInputFormat in our implementation, and we would like to thank the authors for having opened sources of their project.

Chapter 3. Proposed Approach

3.1 Approach Overview

Our system provides a solution for the network flow profiling in the context of P2P applications, be it legal or botnet P2P applications. Given a fine-grained flow, the detection system can differentiate between P2P flow and a normal flow. In case of P2P flow, the system is able to categorize the flow into one of the known P2P applications (e.g. Skype, Storm), or categorize it as a new unknown type of P2P application.

Furthermore, implementation on Hadoop gives the system underlying scalability and fault tolerance capabilities. The overall architecture of the system consists of three main modules shown in Figure 6.

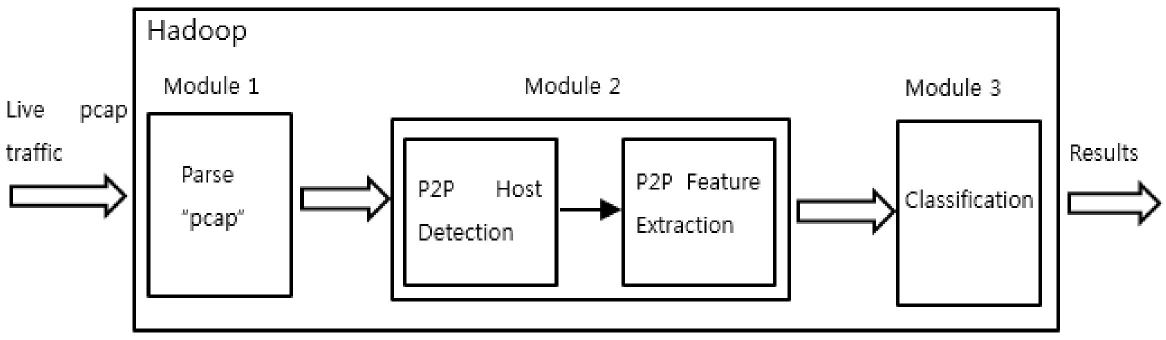


Figure 6. Overview of the system architecture.

The purpose of Module 1 is to parse raw *pcap* files directly from HDFS in parallel. Note that previous studies [37, 38, 39] using Hadoop for botnet detection and generally for traffic analysis utilized traditional *pcap* parsing libraries [40, 48] based on *libpcap* library [47]. However, the mentioned parsing libraries are bound to the resources of the running host machine and cannot scale as Hadoop jobs. The only way the speed of libpcap-based parsers would be comparable to the counterpart in Hadoop is by partitioning input *pcap* files and running multiple parsing processes in parallel. However, even in this case, the resources are bound to the running host, and processing large *pcap* inputs would fail in this case. Note that this module is adopted from the work in [43] where they developed binary input format, namely PcapInputFormat, for processing pcap files from HDFS on Hadoop. Moreover, this adds much flexibility to the system since we can access any fields of the network packets directly from the Hadoop framework without any intermediaries. To our best knowledge, this is the first application of *pcap input format* [43] in the context of botnet detection systems.

Further, main logic of our detection system before entering classification stage is implemented in Module 2. This module itself consists of two sub-modules: P2P host detection sub-module (Module 2.1) and P2P feature extraction sub-module (Module 2.2). The main purpose of Module 2.1 is detecting hosts that appear to be running P2P applications of any type.

A set of features (F_H) used for detecting P2P hosts is described in details in the next section (section #). Another set of features (Fc) have been utilized for further classification. Lastly, classification module (Module 3) on Mahout using random forests has been built.

A number of features differentiate our system from previously existing systems. First of all, it is completely implemented on Hadoop and does not require any third party frameworks that may degrade performance. To best our knowledge, this is first P2P traffic profiling system implemented on Hadoop. Furthermore, Module 2 where most of computation is concentrated is reduced to only one MapReduce job compared to two jobs in similar settings for traffic analysis [43, 44].

3.2 P2P host Detection and Extraction of Flow Intervals

Section 3.2 describes overall idea behind feature selection and detailed architecture of Module 2 where extraction of all the features was implemented. Insight behind the feature set FH for P2P host detection (Module 2.1) is described in Section 3.2.1. Furthermore, flows are divided into flow intervals with the time windows of 10 to 60 minutes. Then, new set of features Fc are extracted (Module 2.2) with the description given in Section 3.2.2.

3.2.1 P2P Host Detection Feature Engineering

Detecting hosts suspicious of running P2P applications is essential before moving into analyzing each flow in details. On this stage, it is possible to drastically reduce the input size so that further analysis can be faster. In comparison to feature dimension reduction techniques [49], this can be considered an orthogonal approach by reduction of input records size. Thus, in order to differentiate P2P applications from normal user behavior (e.g. browsing, file downloads), we consider a number of features listed below.

Failed Connections. Normally, P2P applications expose higher number of failed connections due to some peers going offline and some new peers joining the overlay network. This dynamic behavior of peers is referred as the peer churn [24] phenomenon. We consider as failed any TCP or UDP flow with outgoing packet but no response packet, and a TCP flow with a reset packet.

Unresolved connections. DNS utilization behavior of P2P applications is different from the one of normal traffic [25]. Hosts running P2P applications resolve the IP list from the peers as opposed to DNS query. Thus we consider the number of DNS queries (answers) sent (received) as well as whether the flow have been previously resolved from the DNS answer.

Destination subnet diversity. Another distinction of P2P traffic from normal Internet traffic is the diversity of destination hosts. Usually hosts communicating with bot are scattered around numerous subnets separated geographically. On the other hand, normal user behavior usually exhibits locality. According to mentioned behavior, we extracted the following two features: number of distinct IPs contacted by the host, and the number of different /16 prefix subnets connected by the host.

Thus, Table 1 summarizes the features used for detecting P2P hosts in Module 2.1.

Table 1. Features for P2P hosts detection.

Feature(F_H)	Description	Type	Granularity
fcon	Failed - no reply packets	Binary	Per flow
nrst	# of reset packets	Numerical	Per host
rslvd	Resolved IP	Binary	Per flow
ndns	# of DNS packets exchanged	Numerical	Per host
nddsub	# of distinct destination subnets	Numerical	Per host
nddip	# of distinct destination IPs	Numerical	Per host

During detection certain thresholds are set for the numerical types of features. More detailed information about the threshold settings will be given in the Section 4.1.

3.2.2 Classification Feature Engineering

Once P2P hosts are detected, we extract flow intervals with the related features. Those features are selected specifically to describe the communication patterns inherent to different types of P2P applications. Note that we aim to classify the control flow communications of the P2P applications rather than data flow communications. The reason is that control flow communications are usually correlated with the P2P protocol in usage, whereas data flow communications usually correlated with random user behavior. Moreover, control flow communications exhibit persistent behavior over long periods of time compared to rather chaotic data usage patterns of the users.

Thus, we have selected two major types of features describing either size of data transferred per unit of time or timing of packets. The size of packets matters since control flow packets are normally of certain size. Furthermore, some control flow packets (*e.g.* keep-alive messages) are exchanged periodically for maintaining P2P overlay network. More detailed information for classification features is given in Table 2.

Table 2. Classification features.

Feature(F_C)	Description/Direction	Type
bcF	Byte count/forward	Numerical
pcF	Packet count/forward	Numerical
avBytesF	Average bytes per packet/forward	Numerical
bcB	Byte count/backward	Numerical
pcB	Packet count/backward	Numerical

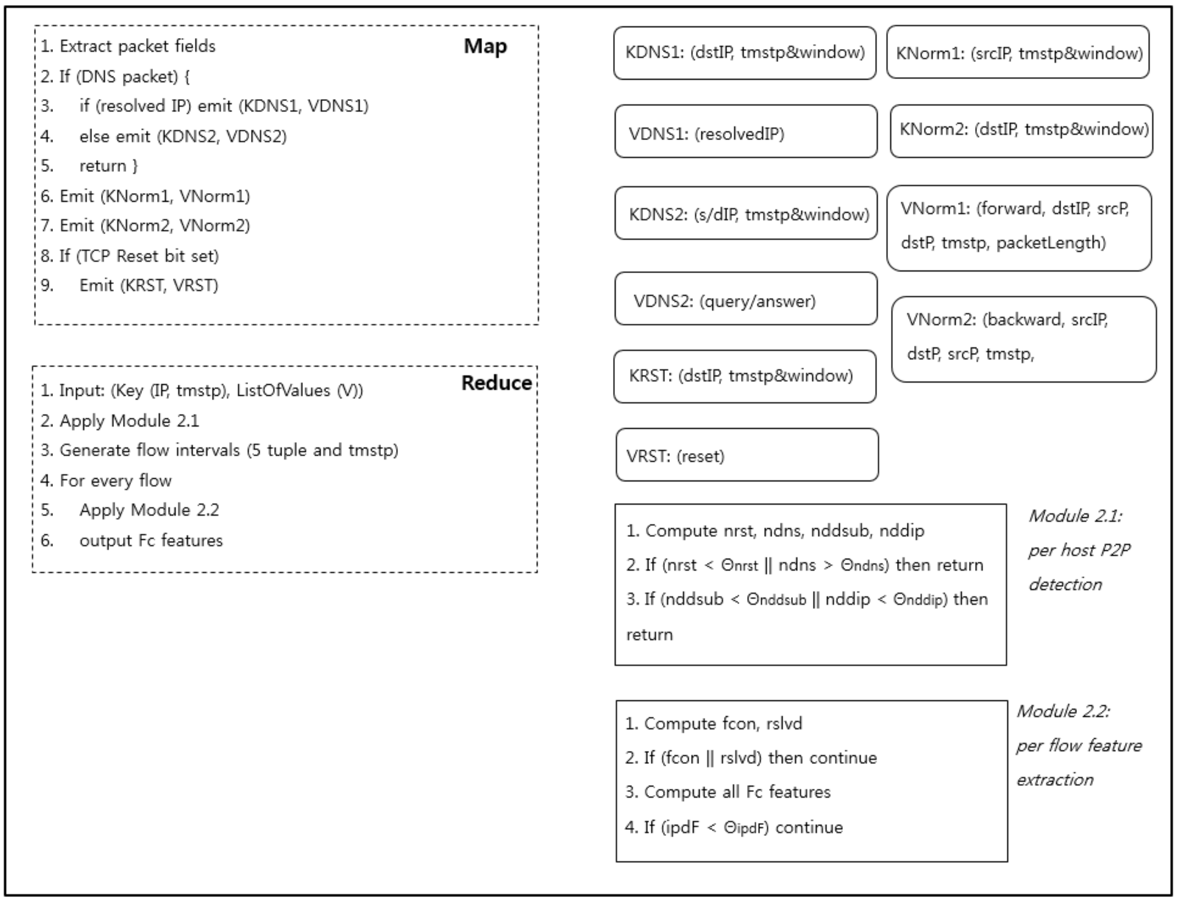
avBytesB	Average bytes per packet/backward	Numerical
ipdF	Average inter packet delay/forward	Numerical
ipdB	Average inter packet delay/backward	Numerical
varipdF	Variance of inter packet delay/forward	Numerical
varipdB	Variance of inter packet delay/backward	Numerical

Inter packet delay (IPD) and its derivatives are one of the most distinctive features for the P2P control flows since they're exchanged periodically with some interval. Moreover, those intervals are different for different types of P2P protocols. Thus, if we have N packets sent during a window of W minutes, then we will have N – 1 IPD intervals, $ipdF_i$ i = 1... N-1. The average of those N -1 intervals would be ipdF, which is calculated as $ipdF = \sum ipdF_i / (N-1)$.

Variance of IPD intervals is another feature that is correlated with types of packets exchanged. For example, in case of variance close to zero, we may imply that packet IPD intervals are relatively stable. In turn, it could be inferred that the corresponding flow interval belongs to the P2P keep-alive messages. Further, we will train random forests classifier to make more sense out of these features.

3.2.3 Implementation on Hadoop

This section gives detailed overview of the design and implementation for Module 2 (code in Appendix) where most of the logic regarding traffic processing is implemented. Thus detailed logical architecture of Module 2 is given in Figure 7 followed by line-wise explanation for Map and Reduce stages.



Map

```
1. Extract packet fields
2. If (DNS packet) {
3.     if (resolved IP) emit (KDNS1, VDNS1)
4.     else emit (KDNS2, VDNS2)
5.     return }
6. Emit (KNorm1, VNorm1)
7. Emit (KNorm2, VNorm2)
8. If (TCP Reset bit set)
9.     Emit (KRST, VRST)
```

Reduce

```
1. Input: (Key (IP, tmstp), ListOfValues (V))
2. Apply Module 2.1
3. Generate flow intervals (5 tuple and tmstp)
4. For every flow
5.     Apply Module 2.2
6.     output Fc features
```

KDNS1: (dstIP, tmstp&window)

VDNS1: (resolvedIP)

KDNS2: (s/dIP, tmstp&window)

VDNS2: (query/answer)

KRST: (dstIP, tmstp&window)

VRST: (reset)

KNorm1: (srcIP, tmstp&window)

KNorm2: (dstIP, tmstp&window)

VNorm1: (forward, dstIP, srcP, dstP, tmstp, packetLength)

VNorm2: (backward, srcIP, dstP, srcP, tmstp,

Module 2.1: per host P2P detection

```
1. Compute nrst, ndns, nddsub, nddip
2. If (nrst < $\Theta_{nrst}$ || ndns > $\Theta_{ndns}$) then return
3. If (nddsub < $\Theta_{nddsub}$ || nddip < $\Theta_{nddip}$) then
return
```

Module 2.2: per flow feature extraction

```
1. Compute fcon, rslvd
2. If (fcon || rslvd) then continue
3. Compute all Fc features
4. If (ipdF < $\Theta_{ipdF}$) continue
```

Figure 7. Detailed logical architecture of Module 2.

Map stage

1. Extraction of the packet fields from the parsed *PcapInputFormat* object coming from previous module

2. Check whether the packet is DNS packet.

3. If host with IP1 receives packet containing resolved IP (e.g. DNS A type answer packet),

then emit a pair with key of IP1 and value of resolved IP.

4. In case of different type of DNS packet (e.g. query, PTR type), let the Reduce stage know about packet for accounting purposes.

5. Return once the DNS packet is processed.

6-7. Since flow is characterized by communications in forward as well as backward direction, line 6 sends extracted packet fields for computing F_c features in the Reduce stage in the forward flow direction. Line 7 serves the same purpose as line 6 with the backward direction of the flow.

8-9. These lines serve as a notification to the Reduce stage regarding the “reset” bit in the TCP packet.

Reduce stage.

1. After emitting from Map stage, the values are sorted according to the keys, and all the values having same key are grouped together. Thus for the key pair of (IP, tmstp), the list of values will include all packets that have either source or destination IP equal the IP in the key pair. Moreover, the timestamps of the packets should be in the interval of tmstp ~ tmstp + window.

2. Module 2.1 implements coarse-grained P2P host detection.

    2.1. Compute four host-based features mentioned in Table 1.

    2.2 – 2.3. Compare the four features with the corresponding thresholds

3-4. Generate flow intervals (source IP, source port, destination IP, destination port, protocol, timestamp) and for every flow do the following.

5. Module 2.2 goes through fine-grained flow-based analysis. For the flows that pass through Module 2.2 final features (F_c) are extracted.

5.1 Compute two flow-based features from Table 1.

5.2 If either of the conditions is true then discard the flow and move on.

5.3 Compute a feature set F_c from Table 2.

5.4 If the value of ipdF feature is smaller than the threshold value then discard the flow and move on.

6. Output newly computed feature set (F_c) for the given flow.

Finally we have the input feature vectors for the classification module.

3.3 Random Forests on Apache Mahout

We have chosen random forests [53] as a main classification algorithm in our framework. Random forests can be categorized as an ensemble method with decision trees performing the role of an ensemble node. Note that random forests have inherent distributed computation property due to possibility of building different decision trees of the forest separately. Figure 8 gives schematic view of random forest.

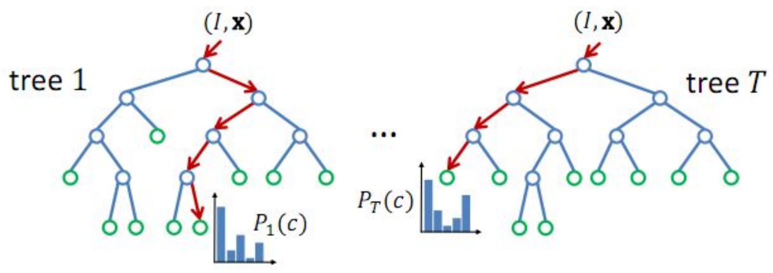


Figure 8. Schematic view of random forest.

The random forest in Figure 8 includes T decision trees. Each node i of the decision tree has the probability distribution Pi(c) over the classes c of data. At the end the probabilities obtained from every tree are averaged for every input record. We have default value of T = 100 in our configuration of random forest.

Note that random forests are implemented as a library in Apache Mahout [42] project by Apache Software Foundation. Mahout is a machine learning framework on top of Hadoop, and it includes a number of learning algorithms. Random forests best fit our needs among the few algorithms implemented in Apache Mahout.

3.4 Benchmark Datasets

Our first dataset, namely ISOT dataset, was created by Information Security and Object Technology (ISOT) research lab at the University of Victoria [32]. Basically, this is a mix of several existing open (malicious and non-malicious) datasets. The malicious traffic is obtained from French chapter of honeynet project [50] and includes Storm, Waledac, and Zeus botnets. Storm botnet had its peak in 2007 - 2008 with more than a million infected bots. In addition, Waledac was considered as the successor of Storm with well distributed P2P style communication protocol. Unlike overnet used by Storm, Waledac utilizes HTTP communication and fast-flux DNS network.

Non-malicious traffic was collected from two sources. One was obtained from the Traffic Lab at Ericsson Research in Hungary [51] (everyday usage traffic). This traffic was integrated with second dataset, which is built by Lawrence Berkeley National Lab (LBNL) [52]. This combination is important since Ericsson Lab dataset includes general traffic from a variety of applications as well as HTTP web browsing, World of Warcraft traffic, and traffic from Azureus bittorent client. On the other hand, LNBL traffic comes from a medium-sized enterprise network and consists of five large datasets. In total, ISOT dataset contains 11.4 GB of Wireshark *pcap* format network traces.

Additionally we utilized second dataset that was used for the benchmarking of PeerRush [36] project. It contains one-month traffic from the testbed set in the Georgia Tech university. It includes traffic from the normal P2P applications such as Skype, uTorrent, and Vuze.

The following Table 3 gives detailed information regarding normal P2P applications.

Table 3. Traffic information regarding normal P2P applications

Type of P2P applicaton	# of hosts	Duration	Traffic volume
Skype	7	1 month	6.1 Gb
eMule	2	1 week	19 Gb
Vuze	2	1 week	9.1 Gb
FrostWire	2	1 week	5.5 Gb
uTorrent	2	1 week	26 Gb

This traffic was mixed with ISOT traffic using TCPReplay tool [54]. Thus the timestamps of the packets were changed as if they were running in the same network at the same time. Due to inherent characteristics of each application, the volumes of collected traffic differ as well. For example, since uTorrent is more data-intensive application compared to Skype, we have more traffic generated by uTorrent in smaller amount of time.

3.4 Cluster Configuration

For the sake of a realistic evaluation, we have set up the 5-node Hadoop cluster in our laboratory environment. The cluster includes one master node and four slave nodes.

System configurations on all the nodes are set as follows.

OS: Ubuntu 14.04 LTS 64-bit.

Kernel version: Linux 3.11.0-23-generic.

Hadoop version: Apache Hadoop 1.2.1.

Mahout version: Apache Mahout 0.9.

Hardware configurations of the nodes are as follows.

Master node: 64 Gb RAM, 24 Intel Xeon CPU E5-2620 @ 2.00GHz, 1Tb HDD.

Slave node 1: 64 Gb RAM, 32 Intel Xeon CPU E5-2620 @ 2.60GHz, 1Tb HDD.

Slave node 2: 16 Gb RAM, 8 Intel Core i7-4770 CPU @ 3.40 GHz, 500 Gb HDD.

Slave node 3: 8 Gb RAM, 4 Intel Core i5-760 CPU @ 2.80 GHz, 300 Gb HDD.

Slave node 4: 8 Gb RAM, 4 Intel Core i5-760 CPU @ 2.80 GHz, 300 Gb HDD.

As you can see, the nodes are heterogeneous from hardware perspective. However, since Hadoop is currently hardware-unaware, we couldn't utilize all hardware with maximum effectiveness. This performance improvement will be considered in our future work.

Chapter 4. Results and Discussion

4.1 P2P Host Detection Results

Due to having two stages with the P2P host detection logic (Module 2) followed by flow-based classification logic (Module3), we'll present the results for each stage separately. Thus, Table 4 presents the P2P host detection results.

Table 4. P2P host detection results.

Type of P2P application	Hosts detected/infected	Extracted flow intervals (records)	Extraction time
Skype	6/7	5832	2 min 37 s
eMule	2/2	11843	4 min 14 s
Vuze	2/2	8421	2 min 53 s
FrostWire	2/2	6182	1 min 52 s
uTorrent	2/2	9245	5 min 11 s
Storm botnet	13/13	2866	1 min 27 s
Waledac botnet	3/3	10712	24 s
Zeus botnet	1/1	6075	36 s
Total	31/32 (96.9%)	61176	

In Module 2, we detect all kinds of P2P hosts. Detection includes legitimate as well as malicious P2P hosts. The results of this stage have only one host running Skype not detected as P2P (false negative). Other P2P hosts are detected with 100% accuracy. Thus, overall accuracy of this stage is 96.9% (31 out of 32 hosts).

Third column of Table 4 represents the flow intervals extracted from the detected hosts. The numbers of extracted flow intervals differ although the periods of monitoring are the same. This can be explained by different P2P protocol configurations of each P2P application. For example, some P2P applications may have a period of 10 minute for heart-beat messages, whereas others may have heart- beat message's period set to 45 minutes. In this case, the latter P2P application will have less flow intervals extracted.

Furthermore Table 5 represents our threshold settings during implementation.

Table 5. Our threshold settings.

Threshold	Corresponding feature	Threshold value
Θ_{nrst}	# of reset packets	> 0
Θ_{ndns}	# of DNS packets exchanged	< 5
Θ_{nddsub}	# of distinct destination subnets	>=100
Θ_{nddip}	# of distinct destination IPs	>=600

Θ_{nrst} represents the number of packets with reset bit set received by the given host during given time interval (default 10 minutes). Θ_{ndns} represents the number of DNS packets exchanged during the same time interval. Θ_{nddsub} and Θ_{nddip} represent the number of distinct destination subnets and IPs, respectively. As we can see, P2P hosts are contacting with more than 600 distinct hosts from more than 100 distinct subnets. These results prove our observation regarding the architecture of P2P networks. Note that these thresholds are targeted to be set by network administrator.

Figure 9 represents dependency of the processing time of Module 2 with varying cluster sizes. It was benchmarked on a representative 4.8 Gb of pcap traffic.

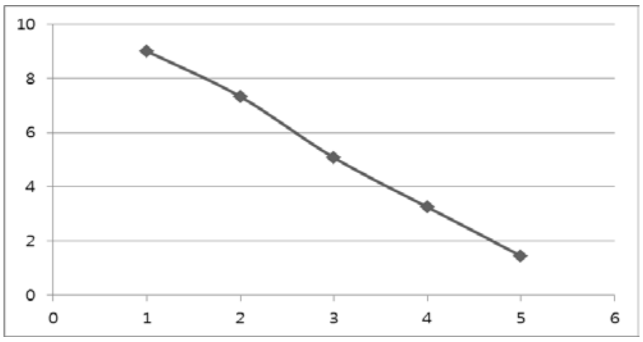


Figure 9. Dependency of processing time from the size of cluster.

Thus we were able to process 4.8 Gb of traffic on a 5-node cluster in 1 minute and 27 seconds. Furthermore, as you can see, the processing time decreases quite rapidly with the increase in cluster size. This is one of the advantages of such a horizontal scaling. Note that the number of Hadoop reduce tasks in this case is set to 10.

4.2 P2P Flow Interval Classification Results

Once the P2P hosts are detected and flow intervals are extracted, we move on to the next stage of training random forest ensemble classifier. For the evaluation we used 10-fold cross validation technique. The building time of classifier was 5 minutes and 47 seconds. The detailed classification results are given in Table 6.

Table 6. Flow intervals classification results.

Type of P2P application	Total flow intervals	Correctly classified (accuracy %)	False positives (rate %)
Skype	5832	5496 (94.24%)	236 (4.04%)
eMule	11843	11538 (97.43%)	378 (3.19%)
Vuze	8421	8008 (95.10%)	312 (3.70%)
FrostWire	6182	6043 (97.75%)	294 (4.75%)
uTorrent	9245	8899 (96.26%)	338 (3.65%)
Storm botnet	2866	2763 (98.11%)	176 (6.14%)
Waledac botnet	10712	10353 (96.65%)	359 (3.35%)

Zeus botnet	6075	5731 (94.33%)	252 (4.15%)
Total	61176	58831 (96.1%)	2194 (3.58%)

As you can see, the average accuracy of our classifier is 96.1% with the corresponding 3.58% of false positive rate. False negatives are not possible in our case (~0%) since we separate normal hosts from P2P hosts in the Module 2.

The comparison of our performance with the previous state-of-the-art systems is given in Table 7. These numbers are given for the processing of representative 1 Gb of pcap traffic in the similar cluster settings.

Table 7. Processing times of each system component.

Method	Feature extraction	Mahout classification	Total	
Our approach	24s	7s	31s	
K. Singh et al.	57s	6s	63s	

The training times are similar in both cases due to usage of Mahout. Note that our method can be used as near-online detection method with a delay of 15~25 minutes depending on the cluster and network settings.

Chapter 5. Concluding Remarks

5.1 Conclusion

Thus we have developed a P2P botnet detection and classification system with its inherent scalability properties. Our system can handle huge traffic by horizontal scaling with the number of machines in the cluster. This was possible due to the development in the Hadoop distributed computing framework.

Furthermore, we have decreased to minimum the number of Hadoop MapReduce jobs required for feature extraction. This gives an advantage of faster processing without additional read/write overheads compared to having more than one job. Moreover, current accuracy of the system is 96.1%, and it can be further improved by using additional features. This brings our system to the near-front of state-of-the-art detection and classification methods. Moreover, our system showed faster performance compared to the state-of-the-art detection methods on similar cluster settings.

5.2 Limitations and Future Work

First of all, we need more in depth analysis of our system in the context of different cluster configurations. Currently our Hadoop cluster is hardware unaware, meaning we don't utilize all the hardware power of our cluster. One of the future solutions may include setting multiple virtual machine on each physical node.

Further in the future, we may add dynamic feature selection algorithm in case of feature dimension gets high. Moreover, implementing similar idea in the context of Software Defined Network (SDN) seems a reasonable next step since SDN provides flow-level view of the system without laborious work of parsing raw network traffic.

References

[1] Stone-Gross, B., Holz, T., Stringhini, G., & Vigna, G. (2011, March). The underground economy of spam: A botmaster’s perspective of coordinating large-scale spam campaigns. In USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET).

[2] Ramsbrock, D., Wang, X., & Jiang, X. (2008, January). A first step towards live botmaster traceback. In Recent Advances in Intrusion Detection (pp. 59-77). Springer Berlin Heidelberg.

[3] Bauer J.M., van Eeten M.J.G., Wu Y., ITU Study on the Financial Aspects of Network Security: Malware and Spam, Technical Report, ITU – International Telecommunication Union, 2008.

[4] Messmer E., “America’s 10 most wanted botnets”, Damballa, Atlanta, GA, 2009. http://www.networkworld.com/news/2009/072209-botnets.html, Accessed: November 2013

[5] Bacher, P., Holz, T., Kotter, M., & Wicherski, G. (2005). Know your enemy: Tracking botnets.

[6] T. Micro, Worm SDBot, 2003 < http://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/SDBOT >.

[7] Chiang, K., & Lloyd, L. (2007, April). A case study of the rustock rootkit and spam bot. In The First Workshop in Understanding Botnets (Vol. 20).

[8] Daswani, N., & Stoppelman, M. (2007, April). The anatomy of Clickbot. A. In Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets (pp. 11-11). USENIX Association.

[9] Maymounkov, P., & Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In Peer-to-Peer Systems (pp. 53-65). Springer Berlin Heidelberg.

[10] Grizzard, J. B., Sharma, V., Nunnery, C., Kang, B. B., & Dagon, D. (2007, April). Peer-to-peer botnets: Overview and case study. In Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets (pp. 1-1).

[11] Holz, T., Steiner, M., Dahl, F., Biersack, E., & Freiling, F. C. (2008). Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. LEET, 8(1), 1-9.

[12] Wang, P., Sparks, S., & Zou, C. C. (2010). An advanced hybrid peer-to-peer botnet. Dependable and Secure Computing, IEEE Transactions on, 7(2), 113-127.

[13] Silva, S. S., Silva, R. M., Pinto, R. C., & Salles, R. M. (2013). Botnets: A survey. Computer Networks, 57(2), 378-403.

[14] Zhu, Z., Lu, G., Chen, Y., Fu, Z., Roberts, P., & Han, K. (2008, July). Botnet research survey. In Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International (pp. 967-972). IEEE.

[15] Caballero, J., Grier, C., Kreibich, C., & Paxson, V. (2011, August). Measuring Pay-per-Install: The Commoditization of Malware Distribution. In USENIX Security Symposium.

[16] Feily, M., Shahrestani, A., & Ramadass, S. (2009, June). A survey of botnet and botnet detection. In Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on (pp. 268-273). IEEE.

[17] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

[18] Hadoop: Open source implementation of MapReduce. < http://hadoop.apache.org/ >

[19] MapReduce on Wikipedia. < http://en.wikipedia.org/wiki/MapReduce >

[20] Forrest, S., Hofmeyr, S. A., Somayaji, A., & Longstaff, T. A. (1996, May). A sense of self for unix processes. In Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on (pp. 120-128). IEEE.

[21] Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., & Kirda, E. (2010, October). AccessMiner: using system-centric models for malware protection. In Proceedings of the 17th ACM conference on Computer and communications security (pp. 399-412). ACM.

[22] Sommer, R., & Paxson, V. (2010, May). Outside the closed world: On using machine learning for network intrusion detection. In Security and Privacy (SP), 2010 IEEE Symposium on (pp. 305-316). IEEE.

[23] Roesch, M. (1999, November). Snort: Lightweight Intrusion Detection for Networks. In LISA (Vol. 99, pp. 229-238).

[24] Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. Computer networks, 31(23), 2435-2463.

[25] Mehra, P. (2012). A brief study and comparison of snort and bro open source network intrusion detection systems.

[26] Jamshed, M. A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., ... & Park, K. (2012, October). Kargus: a highly-scalable software-based intrusion detection system. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 317-328). ACM.

[27] Goebel, J., & Holz, T. (2007, April). Rishi: Identify bot contaminated hosts by irc nickname evaluation. In Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets (pp. 8-8).

[28] Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., ... & Vigna, G. (2009, November). Your botnet is my botnet: analysis of a botnet takeover. In Proceedings of the 16th ACM conference on Computer and communications security (pp. 635-647). ACM.

[29] Gu, G., Porras, P. A., Yegneswaran, V., Fong, M. W., & Lee, W. (2007, August). BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In USENIX Security (Vol. 7, pp. 1-16).

[30] Gu, G., Perdisci, R., Zhang, J., & Lee, W. (2008, July). BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection. In USENIX Security Symposium (pp. 139-154).

[31] Wang, B., Li, Z., Tu, H., & Ma, J. (2009, March). Measuring Peer-to-Peer botnets using control flow stability. In Availability, Reliability and Security, 2009. ARES'09. International Conference on (pp. 663-669). IEEE.

[32] Zhao, D., Traore, I., Sayed, B., Lu, W., Saad, S., Ghorbani, A., & Garant, D. (2013). Botnet detection based on traffic behavior analysis and flow intervals. Computers & Security, 39, 2-16.

[33] Rossow, C., & Dietrich, C. J. (2013). Provex: Detecting botnets with encrypted command and control channels. In Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 21-40). Springer Berlin Heidelberg.

[34] Zhang, J., Perdisci, R., Lee, W., Luo, X., & Sarfraz, U. (2014). Building a scalable system for stealthy p2p-botnet detection.

[35] Zhang, T., Ramakrishnan, R., & Livny, M. (1996, June). BIRCH: an efficient data clustering method for very large databases. In ACM SIGMOD Record (Vol. 25, No. 2, pp. 103-114). ACM.

[36] Rahbarinia, B., Perdisci, R., Lanzi, A., & Li, K. (2014). Peerrush: Mining for unwanted p2p traffic. Journal of Information Security and Applications.

[37] Zhao, Y., Xie, Y., Yu, F., Ke, Q., Yu, Y., Chen, Y., & Gillum, E. (2009, April). BotGraph: Large Scale Spamming Botnet Detection. In NSDI (Vol. 9, pp. 321-334).

[38] Francois, J., Wang, S., Bronzi, W., State, R., & Engel, T. (2011, November). BotCloud: detecting botnets using MapReduce. In Information Forensics and Security (WIFS), 2011 IEEE International Workshop on (pp. 1-6). IEEE.

[39] Singh, K., Guntuku, S. C., Thakur, A., & Hota, C. (2014). Big Data Analytics framework for Peer-to-Peer Botnet detection using Random Forests. Information Sciences, 278, 488-497.

[40] Tshark . < https://www.wireshark.org/docs/man-pages/tshark.html >

[41] Apache Hive, < https://hive.apache.org/ >

[42] Apache Mahout, < http://mahout.apache.org/ >

[43] Lee, Y., Kang, W., & Lee, Y. (2011). A hadoop-based packet trace processing tool (pp. 51-63). Springer Berlin Heidelberg.

[44] Lee, Y., & Lee, Y. (2013). Toward scalable internet traffic measurement and analysis with hadoop. ACM SIGCOMM Computer Communication Review, 43(1), 5-13.

[45] CAIDA CoralReef Software Suite, < http://www.caida.org/tools/measurement/coralreef/ >

[46] RIPE Hadoop Pcap, < https://labs.ripe.net/Members/wnagele/large-scale-pcap-data-analysis-using-apache-hadoop >

[47] Packet capture library (libpcap), < http://wiki.wireshark.org/libpcap >.

[48] Wireshark, < https://www.wireshark.org/ >

[49] Fodor, I. K. (2002). A survey of dimension reduction techniques.

[50] French Chapter of Honeynet project: http://www.honeynet.org/chapters/france. Accessed: November, 2014

[51] Szabó, G., Orincsay, D., Malomsoky, S., & Szabó, I. (2008). On the validation of traffic classification algorithms. In Passive and Active Network Measurement (pp. 72-81). Springer Berlin Heidelberg.

[52] LBNL Enterprise Trace Repository. http://www.icir.org/enterprise-tracing. Accessed: November 2014

[53] Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.

[54] TCPReplay Tool, <http://tcpreplay.synfin.net/>, November, 2014.

Appendix

Source Code on https://github.com/khalid-h/P2PBotnetDetection1.3.git

FlowAnalyzer.java:

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Formatter;
import java.util.Random;
import java.util.Set;
import java.util.StringTokenizer;
import java.lang.*;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextOutputFormat;
import p3.common.lib.BinaryUtils;
import p3.common.lib.BitAdder;
import p3.common.lib.Bytes;
import p3.common.lib.CommonData;
import p3.hadoop.common.pcap.lib.ExtendedBytesWritable;
import p3.hadoop.common.pcap.lib.PcapRec;
import p3.hadoop.mapred.BinaryInputFormat;
import p3.hadoop.mapred.BinaryOutputFormat;
import p3.hadoop.mapred.PcapInputFormat;
public class FlowAnalyzer {
private static final int MIN_PKT_SIZE = 42;
private static final int FLOW_RECORD_SIZE = 17+PcapRec.LEN_VAL3;//+5;
public JobConf conf;
private static byte[] flow_forward = {0x00};
private static byte[] flow_backward = {0x01};
private static int dns_threshold = 0;
private static int reset_threshold = 0;
private static int diverseSubnet_threshold = 10;
public FlowAnalyzer(){
    this.conf = new JobConf();
}
public FlowAnalyzer(JobConf conf){
    this.conf = conf;
}
public static String bytesToHexString(byte[] bytes) {
    StringBuilder sb = new StringBuilder(bytes.length * 2);
    Formatter formatter = new Formatter(sb);
    for (byte b : bytes) {
        formatter.format("%02d", b);
    }
    return sb.toString();
}
/********************************************
    FLOW GEN function
 ********************************************/
public static class Map_FlowGen extends MapReduceBase
implements Mapper<LongWritable, BytesWritable, BytesWritable, BytesWrita-
ble>{
static int interval;
//int dnsTTL = 3600;
int tmstp_decimal = 4;          // meaning 4 digits after decimal point
int subnetThreshold = 100;
public void configure(JobConf conf){
interval = conf.getInt("pcap.record.rate.windowSize", 3600);
}
public static int bytesCompareTo(byte[] a, byte[] b, int length) {
    int res;
    int b1, b2;
    for (int i = 1; i <= length; i++){
        b1 = BinaryUtils.byteToInt(a, i);
        b2 = BinaryUtils.byteToInt(b, i);
         if( b1 > b2 ){
            // first is greater
            return 1;
          }
          else if( b1 < b2 ){
            // second is greater
            return -1;
          }
    }
    return 0;
}
/* returns either 2 bytes ( 0 - query, different from 0 - response type)
 *              or 8 bytes (4 bytes - TTL, 4 bytes - IP) 2 bytes for type
are removed for now since returns IP only for A type */
public static byte[] parse_dns(byte[] packet) {
    byte[] temp1 = {0x00};
    byte[] temp2 = {0x00, 0x00};
    byte[] temp4 = {0x00, 0x00, 0x00, 0x00};
    byte[] malformedReply = {0x00, 0x01};
    byte[] record = {0x00, 0x00};
    int plen = packet.length;
    if (packet.length < 3)
        return malformedReply;
    System.arraycopy(packet, 2, temp1, 0, 1);
    int QR = (int)((byte) (temp1[0] & 0x80) >> 7);
    QR *=QR;
    //int QR;
    if (QR != 0 && QR != 1) {
        System.out.println("Neither query nor answer! check value of
QR!");
        System.exit(1);
    }
    //System.out.println("New packet arrived!");
    //System.out.println("The QR code is as follows: " + QR);
    if (QR == 1) {
        /* dns reply */
        System.arraycopy(packet, 4, temp2, 0, 2);
        int QDCOUNT = Bytes.toInt(temp2);
        System.out.println("Number of questions:\t" + QDCOUNT);
        System.arraycopy(packet, 6, temp2, 0, 2);
        int ANCOUNT = Bytes.toInt(temp2);
        System.out.println("Number of answers:\t" + ANCOUNT);
        int index = 12, iword; /* first byte of query section right after
12 bytes of header section*/
        for (int i = 0; i < QDCOUNT; i++) {
            if (index >= plen - 1) {
                return malformedReply;
            }
            System.arraycopy(packet, index, temp1, 0, 1);
            iword= Bytes.toInt(temp1);
            while (iword != 0){
                System.out.println("\tname label of \t" + iword +
"\tbytes");
                index += iword + 1;
                if (index >= plen - 1) {
                    return malformedReply;
                }
                System.arraycopy(packet, index, temp1, 0, 1);
                iword = Bytes.toInt(temp1);
            }
            index += 5 ;    /* 2 bytes for QTYPE, 2 bytes for QCLASS of
question field + 1 */
        }
        //System.out.println("index is \t" + index + "length is \t" +
packet.length);
        iword = 0;
        int ptr = 0;
        if (ANCOUNT >= 1) {
            //to be deleted
            if (ANCOUNT > 1){
                System.out.println("WARNING! Answer count (ANCOUNT) is
greater than 1. Currently handle only first packet");
            }
            //byte[] nullByte = {0x00};
            //return nullByte;
        //for (int i = 0; i < ANCOUNT; i++) {       only for first answer
field
            System.arraycopy(packet, index, temp1, 0, 1);
            iword = Bytes.toInt(temp1);
            //System.out.println(iword);
            ptr = temp1[0] & 0xC0;
            while (iword != 0 && ptr != 192){
                System.out.println("name label of \t" + iword +
"\tbytes");
                index += iword + 1;
                if (index >= plen - 1) {
                    return malformedReply;
                }
                System.arraycopy(packet, index, temp1, 0, 1);
                ptr = temp1[0] & 0xC0;
                iword = Bytes.toInt(temp1);
            }
            if (ptr == 192) {
                System.arraycopy(packet, index, temp2, 0, 2);
                temp2[0] = (byte) (temp2[0] & 0x3F);        // mask out
(zerofy) first two bits
                int qname_ptr = Bytes.toInt(temp2);
                System.out.println("Qname pointer is :\t" + qname_ptr);
                index += 2;     //if ends by pointer then forward 2 bytes
(since pointer takes 2 bytes)
            } else {
                index ++;       //no pointer, null takes 1 byte
            }
            System.arraycopy(packet, index, temp2, 0, 2);
            int TYPE = Bytes.toInt(temp2);
            System.arraycopy(packet, index, record, 0, 2);
            System.out.println("The RDATA type field is\t" + TYPE);
            // currently only for A type answers. for IPV6 (AAAA) need to
add " || TYPE == 28 "
            if (TYPE == 1) {
                index += 4;         /* move pointer to TTL field of An-
swer section  (skipping CLASS field) */
                System.arraycopy(packet, index, temp4, 0, 4);
                int TTL = Bytes.toInt(temp4);
                System.out.println("TTL of the packet is\t" + TTL);
                index += 4;
                System.arraycopy(packet, index, temp2, 0, 2);
                int RDLENGTH = Bytes.toInt(temp2);
                System.out.println("The length of RDATA field is\t" +
RDLENGTH);
                assert(RDLENGTH==4);
                index += 2;
                byte[] ipv = new byte[RDLENGTH + 4];
                //System.arraycopy(record, 0, ipv, 0, 2);       // TYPE :
bytes 0-1
                System.arraycopy(temp4, 0, ipv, 0, 4);          // TTL :
bytes 0-3
                System.arraycopy(packet, index, ipv, 4, RDLENGTH);  //IP:
bytes 4 - 4 +RDLENGTH (4)
                //System.out.println("A type record with IP:\t" + Common-
Data.longTostrIp(Bytes.toLong(ipv)));
                index += RDLENGTH;
                return ipv;     //change in case of ANCOUNT > 1
            }
        }       //if ANCOUNT >= 1, for more than 1 answer treats same as
1 answer. Can be changed later with uncommenting "for" loop
    }
    return record;
}
public static void dns_handle(byte[] value_bytes, int proto, long
cap_stime, long  cap_stime_mod, OutputCollector<BytesWritable, BytesWrit-
able> output) throws IOException {
    ExtendedBytesWritable new_key = new ExtendedBytesWritable(new
byte[8]);
    ExtendedBytesWritable new_value = new ExtendedBytesWritable(new
byte[4]);
    ExtendedBytesWritable new_value2 = new ExtendedBytesWritable(new
byte[1]);
    ArrayList<byte[]> pair = new ArrayList<byte[]>(2);
    byte[] dnsType = {0x00, 0x00};
    byte[] dnsAType = {0x00, 0x01};
    byte[] vtype_dnsquery = {0x01};
    byte[] vtype_dnsanswer = {0x02};
    byte[] resolvedTTL = new byte[4];
    byte[] resolvedIP = new byte[4];
    byte[] bsip = new byte[4];
    int dnsplen = -1;
    if (proto == PcapRec.UDP) {
        byte[] dns_blen = {0x00, 0x00};
        System.arraycopy(value_bytes, PcapRec.POS_SP + 4, dns_blen, 0,
2);
        int dns_len = Bytes.toInt(dns_blen) - 8; //extracting UDP header
length
        int dns_lenValueBytes = value_bytes.length - (PcapRec.POS_SP +
8);
        /* UDP header starts at position 50 and following DNS header
starts at 58 */
        //System.out.println("Packet capture time" + (cap_stime +
cap_stime_mod));
        byte[] dns_packet = new byte[dns_len];
        //System.out.println("size of value_bytes" +  dns_lenValueBytes +
"size of calculated length" + dns_len);
        System.arraycopy(value_bytes, PcapRec.POS_SP + 8, dns_packet, 0,
Math.min(dns_lenValueBytes, dns_len));
        byte[] dnsInfo = parse_dns(dns_packet);
        dnsplen = dnsInfo.length;
        switch (dnsplen) {
        case 2:
            System.arraycopy(dnsInfo, 0, dnsType, 0, 2);
            break;
        case 8:
            System.arraycopy(dnsAType, 0, dnsType, 0, 2);
            System.arraycopy(dnsInfo, 0, resolvedTTL, 0, 4);
            System.arraycopy(dnsInfo, 4, resolvedIP, 0, 4);
            break;
        default:
            System.out.println("Default case!");
        }
    }
    if (proto == PcapRec.TCP) {
        System.out.println("WARNING!!! Handle DNS packet for TCP");
        return;
        //System.exit(1);
    }
    assert(dnsplen != -1);
    int type = Bytes.toInt(dnsType);
    if (type == 0) {
        // dns query.
        new_key.set(value_bytes, PcapRec.POS_SIP, 0, 4);
        new_key.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4, 4);
        new_value2.set(vtype_dnsquery, 0, 0, 1);
        output.collect(new BytesWritable(new_key.getBytes()),  new
BytesWritable(new_value2.getBytes()));
    } else {
        // dns answer.
        if (dnsplen == 8) {
            new_key.set(value_bytes, PcapRec.POS_DIP, 0, 4);
            /*System.arraycopy(value_bytes, PcapRec.POS_DIP, bsip, 0, 4);
            int res2 = bytesCompareTo(bsip,resolvedIP,4);
            if (res2 > 0) {
                new_key.set(bsip, 0, 0, 4);
                new_key.set(resolvedIP, 0, 4, 4);
            } else if (res2 < 0){
                new_key.set(resolvedIP, 0, 0, 4);
                new_key.set(bsip, 0, 4, 4);
            } else {
```

```
//System.out.println(CommonData.longTostrIp(Bytes.toLong(bsip)));

//System.out.println(CommonData.longTostrIp(Bytes.toLong(resolvedIP)));
                System.out.println("Loopback packet 1");
                //System.exit(0);
                return;
            }*/
            new_key.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4, 4);
            //new_value.set(vtype_dnsanswer, 0, 0, 1);
            new_value.set(resolvedIP, 0, 0, 4);
            //new_value.set(resolvedTTL, 0, 1, 4);
            //new_value.set(BinaryUtils.uIntToBytes(cap_stime_mod), 0, 1,
4);
            //new_key2.set(value_bytes,  PcapRec.POS_DIP, 0, 4);
            //new_key2.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4, 4);
            //new_value2.set(vtype_dnsanswer, 0, 0, 1);
            output.collect(new BytesWritable(new_key.getBytes()),  new
BytesWritable(new_value.getBytes()));
        } else {
            new_key.set(value_bytes,  PcapRec.POS_DIP, 0, 4);
            new_key.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4, 4);
            new_value2.set(vtype_dnsanswer, 0, 0, 1);
            output.collect(new BytesWritable(new_key.getBytes()),  new
BytesWritable(new_value2.getBytes()));
        }
    }
}
    public void map
        (LongWritable key, BytesWritable value,
        OutputCollector<BytesWritable, BytesWritable> output, Reporter
reporter) throws IOException {
        ExtendedBytesWritable new_key_src = new ExtendedBytesWritable(new
byte[8]);
        ExtendedBytesWritable new_key_dst = new ExtendedBytesWritable(new
byte[8]);
        ExtendedBytesWritable new_key_rst = new ExtendedBytesWritable(new
byte[8]);
        //ExtendedBytesWritable new_key3 = new ExtendedBytesWritable(new
byte[8]);
        ExtendedBytesWritable new_value_rst = new ExtendedBytesWrita-
ble(new byte[2]);
        ExtendedBytesWritable value_normal_src = new ExtendedBytesWrita-
ble(new byte[15]);
        ExtendedBytesWritable value_normal_dst = new ExtendedBytesWrita-
ble(new byte[15]);
        ArrayList<byte[]> dns_kvPair = new ArrayList<byte[]>(2);
        byte[] value_bytes = value.getBytes();
        byte[] eth_type = new byte[2];
        byte[] bsrc_port = new byte[2];
        byte[] bdst_port = new byte[2];
        byte[] ip_proto = {0x00};
        byte[] bcap_time = new byte[4];
        byte[] vtype_normal = {0x00};
        byte[] vtype_dnsquery = {0x01};
        byte[] vtype_reset = {0x03, 0x04};
        byte[] bsip = new byte[4];
        byte[] bdip = new byte[4];
        byte[] tcp_flags = new byte[1];
        long src_port, dst_port, cap_stime = 0, cap_stime_mod = 0,
cap_mstime = 0;
```

```
        int proto;
        byte[] temp = new byte[48];
        //System.exit(0);
        if(value_bytes.length<MIN_PKT_SIZE) return;
        System.arraycopy(value_bytes, PcapRec.POS_ETH_TYPE, eth_type, 0,
PcapRec.LEN_ETH_TYPE);
        /* System.out.println(BinaryUtils.byteToInt(eth_type));
        System.out.println("Length of packet is" + value_bytes.length);
        System.out.println(value_bytes);
        */
        if(BinaryUtils.byteToInt(eth_type) != PcapRec.IP_PROTO) {
            System.arraycopy(value_bytes, PcapRec.POS_ETH_TYPE + 2,
eth_type, 0, PcapRec.LEN_ETH_TYPE);
            //System.out.println(String.format("%02X ", eth_type[0]) + "
" + String.format("%02X ", eth_type[1]));
            if (BinaryUtils.byteToInt(eth_type) == PcapRec.IP_PROTO) {
                /* System.out.println(value_bytes.length);
                System.arraycopy(value_bytes, 0, temp, 0, 48);
                StringBuilder sb = new StringBuilder();
                for (int i = 0; i < temp.length; i++) {
                    if (i == 16 || i == 32)
                        sb.append("\n");
                    sb.append(String.format("%02X ", temp[i]));
                }
                System.out.println(sb.toString());
                */
                // deleting 00 00 (2 bytes) in positions 28,29
                byte[] new_value_bytes = new byte[value_bytes.length -
2];
                System.arraycopy(value_bytes, 0, new_value_bytes, 0,
PcapRec.POS_ETH_TYPE);
                System.arraycopy(value_bytes, PcapRec.POS_ETH_TYPE + 2,
new_value_bytes, PcapRec.POS_ETH_TYPE, new_value_bytes.length -
PcapRec.POS_ETH_TYPE);
                value_bytes = new_value_bytes;
                System.arraycopy(value_bytes, PcapRec.POS_ETH_TYPE,
eth_type, 0, PcapRec.LEN_ETH_TYPE);
                //System.exit(0);
            } else {
                System.out.println("Warning! Not IP type packet");
                System.out.println("Type of packet: " + Bina-
ryUtils.byteToInt(eth_type));
                //System.exit(0);
                return;
            }
        }
        System.arraycopy(value_bytes, PcapRec.POS_SIP + PcapRec.LEN_VAL2,
bsrc_port, 0, 2);
        System.arraycopy(value_bytes, PcapRec.POS_SIP + PcapRec.LEN_VAL2
+ PcapRec.LEN_PORT, bdst_port, 0, 2);
        src_port = Bytes.toLong(bsrc_port);
        dst_port = Bytes.toLong(bdst_port);
        System.arraycopy(value_bytes, PcapRec.POS_PT, ip_proto, 0, 1);
        proto = Bytes.toInt(ip_proto);
        System.arraycopy(value_bytes, PcapRec.POS_TSTMP, bcap_time, 0,
4);

        cap_stime = Bytes.toLong(BinaryUtils.flipBO(bcap_time,4));
        cap_stime_mod = cap_stime % interval;
        cap_stime = cap_stime - cap_stime_mod;
        System.arraycopy(value_bytes, PcapRec.POS_TSTMP + 4, bcap_time,
0, 4);
        cap_mstime = Bytes.toLong(BinaryUtils.flipBO(bcap_time,4));
        System.out.println("time after decimals\t" + cap_mstime);
        // computation of decimal points of timestamp
        cap_mstime = Inte-
ger.valueOf(String.valueOf(cap_mstime).substring(0,Math.min(String.valueO
f(cap_mstime).length(), tmstp_decimal))) + cap_stime_mod * (int)
Math.pow(10, tmstp_decimal);
        System.out.println("time before decimals\t" + cap_stime);
        System.out.println("time after modulus\t" + cap_stime_mod);
        System.out.println("time after addition" + cap_mstime);
        System.out.println("time interval\t" + interval);
        //System.exit(0);
        /* Extract dns packet type. For 'A type' packets extract IP field
*/
        if (src_port == 53 || src_port == 5353 || src_port == 5355 ||
dst_port == 53 || dst_port == 5353 || dst_port == 5355){
            /* 5353 - MDNS; 5355 - LLMNR */
            dns_handle(value_bytes, proto, cap_stime, cap_stime_mod, out-
put);
            return;
        }
        System.arraycopy(value_bytes, PcapRec.POS_SIP, bsip, 0, 4);
        System.arraycopy(value_bytes, PcapRec.POS_DIP, bdip, 0, 4);
        System.out.println("source IP\t" + Common-
Data.longTostrIp(Bytes.toLong(bsip)));
        System.out.println("destination IP\t" + Common-
Data.longTostrIp(Bytes.toLong(bdip)));
        /*
        int res = bytesCompareTo(bsip,bdip,4);
        if (res > 0) {
            //bsip is greater than bdip
            new_key.set(bsip, 0, 0, 4);
            new_key.set(bdip, 0, 4, 4);
            // 0 value if larger IP is source - > smaller IP is destina-
tion
            value_normal.set(vtype_normal, 0, 0, 1);
            value_normal.set(value_bytes, PcapRec.POS_SP, 1,
PcapRec.LEN_PORT);
            value_normal.set(value_bytes, PcapRec.POS_DP, 3,
PcapRec.LEN_PORT);
        } else if (res < 0) {
            //bdip is greater than bsip
            new_key.set(bdip, 0, 0, 4);
            new_key.set(bsip, 0, 4, 4);
            // 1 value if larger IP is destination < - smaller IP is
source
            value_normal.set(vtype_dnsquery, 0, 0, 1);
            value_normal.set(value_bytes, PcapRec.POS_DP, 1,
PcapRec.LEN_PORT);
            value_normal.set(value_bytes, PcapRec.POS_SP, 3,
PcapRec.LEN_PORT);
        } else {

//System.out.println(CommonData.longTostrIp(Bytes.toLong(bsip)));

//System.out.println(CommonData.longTostrIp(Bytes.toLong(bdip)));
            System.out.println("Loopback packet 2");
            return;
            //System.exit(0);
        }
        */
        new_key_src.set(value_bytes, PcapRec.POS_SIP, 0, 4);
        new_key_src.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4, 4);
        new_key_dst.set(value_bytes, PcapRec.POS_DIP, 0, 4);
        new_key_dst.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4, 4);
        // for distinct ip and subnets from sender host
        value_normal_src.set(flow_forward, 0, 0, 1);
        value_normal_dst.set(flow_backward, 0, 0, 1);
        value_normal_src.set(value_bytes, PcapRec.POS_DIP, 1, 4);
        value_normal_dst.set(value_bytes, PcapRec.POS_SIP, 1, 4);
        if (src_port > dst_port) {
            value_normal_src.set(value_bytes, PcapRec.POS_SP, 5,
PcapRec.LEN_PORT);
            value_normal_dst.set(value_bytes, PcapRec.POS_SP, 5,
PcapRec.LEN_PORT);
            value_normal_src.set(value_bytes, PcapRec.POS_DP, 7,
PcapRec.LEN_PORT);
            value_normal_dst.set(value_bytes, PcapRec.POS_DP, 7,
PcapRec.LEN_PORT);
        } else {
            value_normal_src.set(value_bytes, PcapRec.POS_DP, 5,
PcapRec.LEN_PORT);
            value_normal_dst.set(value_bytes, PcapRec.POS_DP, 5,
PcapRec.LEN_PORT);
            value_normal_src.set(value_bytes, PcapRec.POS_SP, 7,
PcapRec.LEN_PORT);
            value_normal_dst.set(value_bytes, PcapRec.POS_SP, 7,
PcapRec.LEN_PORT);
        }
        value_normal_src.set(BinaryUtils.uIntToBytes(cap_mstime), 0, 9,
4);
        value_normal_dst.set(BinaryUtils.uIntToBytes(cap_mstime), 0, 9,
4);
        value_normal_src.set(value_bytes, PcapRec.POS_IP_BYTES, 13,
PcapRec.LEN_IP_BYTES);
        value_normal_dst.set(value_bytes, PcapRec.POS_IP_BYTES, 13,
PcapRec.LEN_IP_BYTES);
        // need to set for normal case
        output.collect(new BytesWritable(new_key_src.getBytes()), new
BytesWritable(value_normal_src.getBytes()));
        output.collect(new BytesWritable(new_key_dst.getBytes()), new
BytesWritable(value_normal_dst.getBytes()));
        if (proto == PcapRec.TCP) {
            System.arraycopy(value_bytes, PcapRec.POS_DP + 11, tcp_flags,
0, 1);
            tcp_flags[0] = (byte) (tcp_flags[0] & 0x04);
            if (tcp_flags[0] == 0x04) {
                new_key_rst.set(value_bytes, PcapRec.POS_DIP, 0, 4);
                new_key_rst.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4,
4);
                new_value_rst.set(vtype_reset, 0, 0, 2);
                output.collect(new BytesWritable(new_key_rst.getBytes()),
new BytesWritable(new_value_rst.getBytes()));
            }
        }
        //new_key3.set(bsip, 0, 0, 4);
        //new_key3.set(BinaryUtils.uIntToBytes(cap_stime), 0, 4, 4);
        //new_value3.set(bdip, 0, 0, 4);
        //output.collect(new BytesWritable(new_key3.getBytes()), new
BytesWritable(new_value3.getBytes()));
        //System.out.println(CommonData.longTostrIp(Bytes.toLong(bsip)));
        //System.out.println(CommonData.longTostrIp(Bytes.toLong(bdip)));
    }
}
 public static class Reduce_FlowGen extends MapReduceBase
    implements Reducer<BytesWritable, BytesWritable, Text, Text> {
     ExtendedBytesWritable new_value1 = new ExtendedBytesWritable(new
byte[4]);
        public void reduce(BytesWritable key, Iterator<BytesWritable>
value,
            OutputCollector<Text, Text> output, Reporter reporter)
            throws IOException {
            int vsize = 0;
            byte[] dns_data = new byte[1];
            byte[] dnsResolved_data = new byte[5];
            byte[] normal_data = new byte[15];
            byte[] sIP = new byte[4];
            byte[] dIP = new byte[4];
            byte[] resolvedIP = new byte[4];
            byte[] tempb = new byte[4];
            byte[] bcap_modTime = new byte[4];
            byte[] bSP = new byte[2];
            byte[] bDP = new byte[2];
            byte[] pbytes = new byte[2];
            byte[] bflow_direction = new byte[1];
            BytesWritable data;
            String strSIP = "0.0.0.0", strTuple = "uninitialized",
strFeatures = "", strKeyPair = "", strValue = "";
            String strip1 = "0.0.0.0", strip2 = "0.0.0.0", dip, subnet,
str_resolvedIP;
            //String  delimeter = "\t", delimeter2 = ":";
            String  delimeter = ",", delimeter2 = ",";
            boolean dnsResolved = false, dnsCount = false, normal_packets
= false, hostRelated = false, output_flag = false;
            int dnsQuery_count = 0, dnsAnswer_count = 0, temp,
bytesperPacket = 0, flowDirection = -1, cap_modTime = 0, reset_count = 0;
            int f3fLen, f3bLen, tempipd, localDuration;
            long sumipd = 0;
            long cap_time = 0;
            double ipdf, ipdb, varf, varb;
            HashMap<String, HashMap> features =  new HashMap<String,
HashMap>();
            HashMap<String, ArrayList> tempfset =  new HashMap<String,
ArrayList>();
            HashMap<String, Integer> ipPool = new HashMap<String, Inte-
ger>();
            HashMap<String, Integer> subnetPool = new HashMap<String, In-
teger>();
            HashMap<String, Integer> resolvedIPpool = new HashMap<String,
Integer>();
            ArrayList<Integer> tempFeature = new ArrayList<Integer>();
            ArrayList<Integer> tempFeature2 = new ArrayList<Integer>();
            ArrayList<Integer> tempFeature3f = new ArrayList<Integer>();
            ArrayList<Integer> tempFeature3b = new ArrayList<Integer>();
            ArrayList<BytesWritable> copy_values = new Ar-
rayList<BytesWritable>();
            int fcount = 0;
            while (value.hasNext()) {
                data = value.next();
```

```
                    vsize = data.getLength();
                    System.out.println("size of value is\t" + vsize);
                    //vsize = value.next().getLength();
                    System.out.println(vsize);
                    switch (vsize) {
                       case 1:
                           if (!dnsCount) {
                               dnsCount = true;
                               System.arraycopy(key.getBytes(), 0, sIP, 0,
4);
                               strSIP = Common-
Data.longTostrIp(Bytes.toLong(sIP));
                           }
                           dns_data = data.getBytes();
                           if (dns_data[0] == 0x01) {
                               //System.out.println("dns query from host \t"
+ strSIP);
                               dnsQuery_count++;
                           } else if (dns_data[0] == 0x02) {
                               //System.out.println("dns answer to host \t"
+ strSIP);
                               dnsAnswer_count++;
                           } else {
                               System.out.println("Incorrect dns value");
                           }
                           break;
                       case 2:
                           // count input packets with reset flag set
                           reset_count ++;
                           break;
                       case 4:
                           // flow based
                           // collecting resolved IPs
                           //dnsResolved = true;
                           resolvedIP = data.getBytes();
                           str_resolvedIP = Common-
Data.longTostrIp(Bytes.toLong(resolvedIP));
                           if (!resolvedIPpool.containsKey(str_resolvedIP)){
                               resolvedIPpool.put(str_resolvedIP, 1);
                           }
                           break;
                       case 15:
                           //System.out.println(strTuple);
                           normal_data = data.getBytes();
                           byte[] copy_record = new byte[15];
                           System.arraycopy(normal_data, 0, copy_record, 0,
15);
                           copy_values.add(new BytesWritable(copy_record));
                           System.arraycopy(normal_data, 0, bflow_direction,
0, 1);
                           System.arraycopy(normal_data, 9, bcap_modTime, 0,
4);
                           System.out.println("Case 15");
                           System.out.println("mstime :\t" +
Bytes.toLong(bcap_modTime));
                           System.out.println("flow originally\t" +
Bytes.toInt(bflow_direction));
                           if (bflow_direction[0] == flow_forward[0]) {
                               fcount++;
                               System.arraycopy(normal_data, 1, dIP, 0, 4);
```

```
                            dip = Common-
Data.longTostrIp(Bytes.toLong(dIP));
                            subnet = dip.substring(0, dip.indexOf('.',
dip.indexOf('.') + 1));      //till second "." for /16 prefix
                            if (ipPool.containsKey(dip)) {
                                ipPool.put(dip, ipPool.get(dip) + 1);
                            } else {
                                ipPool.put(dip, 1);
                                if (subnetPool.containsKey(subnet)){
                                    subnetPool.put(subnet, subnet-
Pool.get(subnet) + 1);
                                } else {
                                    subnetPool.put(subnet, 1);
                                }
                            }
                        }
                        break;
                    default:
                        break;
                }
            }
            /* debugging
             * System.out.println("Forward packets count: \t" + fcount);
            System.out.println("Values after copy, before function
call");
            Iterator<BytesWritable> iterValue = copy_values.iterator();
            while (iterValue.hasNext()){
                data = iterValue.next();
                normal_data = data.getBytes();
                //System.arraycopy(normal_data, 0, bflow_direction, 0,
1);
                System.arraycopy(normal_data, 9, bcap_modTime, 0, 4);
                //System.out.println("Case 15");
                System.out.println("mstime2 :\t" +
Bytes.toLong(bcap_modTime));
            }
            */
            // P2P host detection
            int diverse_subnets = subnetPool.size();
            int dnsTotal_count = dnsQuery_count + dnsAnswer_count;
            if (diverse_subnets >= diverseSubnet_threshold &&
dnsTotal_count >= dns_threshold && reset_count >= reset_threshold) {
                String hostAttr = delimeter + Inte-
ger.toString(diverse_subnets) + delimeter + Inte-
ger.toString(dnsTotal_count) + delimeter + Integer.toString(reset_count);
                System.out.println("Host related features:\t" + hos-
tAttr);
                //System.exit(0);
                extract_classifFeatures(key, copy_values.iterator(), out-
put, hostAttr);
            }
        }
    }
 public static void extract_classifFeatures (BytesWritable key, Itera-
tor<BytesWritable> value, OutputCollector<Text, Text> output, String hos-
tAttr) throws IOException {
        BytesWritable data;
        byte[] IP1 = new byte[4];
        byte[] IP2 = new byte[4];
        byte[] bcap_time = new byte[4];
        byte[] bmcap_time = new byte[4];
        byte[] bSP = new byte[2];
        byte[] bDP = new byte[2];
        byte[] normal_data;
        byte[] bflow_direction = new byte[1];
        byte[] pbytes = new byte[2];
        long cap_time = 0, cap_modTime = 0;
        int flowDirection, bytesperPacket = 0;
        HashMap<String, HashMap> features =  new HashMap<String,
HashMap>();
        HashMap<String, ArrayList> tempfset;
        ArrayList<Integer> tempFeature, tempFeature2, tempFeature3f,
tempFeature3b;
        int f3fLen, f3bLen, tempipd, sumipd;
        double ipdf, ipdb, varf, varb;
        String strFlow = "uninitialized", delimeter = "\t", delimeter2 =
":", strFlowInterval = "uninitialized", strValue = "";
        System.arraycopy(key.getBytes(), 0, IP1, 0, 4);
        String str_IP1 = CommonData.longTostrIp(Bytes.toLong(IP1)),
str_IP2;
        System.arraycopy(key.getBytes(), 4, bcap_time, 0, 4);
        cap_time = Bytes.toInt(bcap_time);
        System.out.println("masked timestamp : " + cap_time);
        int count = 0, fcount = 0;
     while (value.hasNext()) {
        data = value.next();
        normal_data = data.getBytes();
        System.out.println("value next in extraction ");
        System.arraycopy(normal_data, 0, bflow_direction, 0, 1);
        System.arraycopy(normal_data, 1, IP2, 0, 4);
        System.arraycopy(normal_data, 5, bSP, 0, 2);
        System.arraycopy(normal_data, 7, bDP, 0, 2);
        System.arraycopy(normal_data, 9, bmcap_time, 0, 4);
        str_IP2 = CommonData.longTostrIp(Bytes.toLong(IP2));
        flowDirection = Bytes.toInt(bflow_direction);
        strFlow = str_IP1 + delimeter + str_IP2 + delimeter +
Bytes.toInt(bSP) + delimeter + Bytes.toInt(bDP);
        //cap_modTime = Bytes.toInt(bmcap_time);
        cap_modTime = Bytes.toLong(bmcap_time);
        //BinaryUtils.byteToInt() or Bytes.toLong()
        System.out.println("exact time : " + cap_modTime);
        System.out.println("exact time2 : " + Bina-
ryUtils.byteToInt(bmcap_time));
        //System.exit(0);
        System.arraycopy(normal_data, 13, pbytes, 0,
PcapRec.LEN_IP_BYTES);
        bytesperPacket = Bytes.toInt(pbytes);
        System.out.println("bytes per packet\t" + bytesperPacket);
        System.out.println("flow direction\t" + flowDirection);
        if (features.containsKey(strFlow)) {
            // change values for features
            tempfset = features.get(strFlow);
            tempFeature = tempfset.get("bytecount");
            tempFeature.set(flowDirection, tempFeature.get(flowDirection)
+ bytesperPacket);
            tempfset.put("bytecount", tempFeature);
            tempFeature = tempfset.get("packetcount");
            tempFeature.set(flowDirection, tempFeature.get(flowDirection)
+ 1);
            tempfset.put("packetcount", tempFeature);
            if (flowDirection == 0) {
                    tempFeature = tempfset.get("IPDForward");
                    tempFeature.add((int)cap_modTime);
                    tempfset.put("IPDForward", tempFeature);
                } else if (flowDirection == 1) {
                    tempFeature = tempfset.get("IPDBackward");
                    tempFeature.add((int)cap_modTime);
                    tempfset.put("IPDBackward", tempFeature);
                } else {
                    System.out.println("bad value for flow direction");
                    System.exit(1);
                }
            features.put(strFlow, tempfset);
        } else {
            // add all starting values for features
            HashMap<String, ArrayList> fset =  new HashMap<String, Ar-
rayList>();
            ArrayList<Integer> flowfeat1 = new ArrayList<Integer>();
            flowfeat1.add(0);
            flowfeat1.add(0);
            flowfeat1.set(flowDirection, bytesperPacket);
            fset.put("bytecount", flowfeat1);
            ArrayList<Integer> flowfeat2 = new ArrayList<Integer>();
            flowfeat2.add(0);
            flowfeat2.add(0);
            flowfeat2.set(flowDirection, 1);
            fset.put("packetcount", flowfeat2);
            ArrayList<Integer> flowfeat3f = new ArrayList<Integer>();
            ArrayList<Integer> flowfeat3b = new ArrayList<Integer>();
            if (flowDirection == 0) {
                flowfeat3f.add((int)cap_modTime);
            } else if (flowDirection == 1) {
                flowfeat3b.add((int)cap_modTime);
            } else {
                System.out.println("bad value for flow direction");
                System.exit(1);
            }
            fset.put("IPDForward", flowfeat3f);
            fset.put("IPDBackward", flowfeat3b);
            features.put(strFlow, fset);
        }
        //System.out.println(strFlow);
        //System.out.println(features.get(strFlow));
        //if (count == 1) System.exit(0);
        count++;
     }
     System.out.println(features);
     System.out.println("Count\t" + count);
     //System.exit(0);
     for (String fkey: features.keySet()){
       strFlowInterval = fkey + delimeter + String.valueOf(cap_time) +
delimeter2;
       tempFeature = (ArrayList<Integer>) fea-
tures.get(fkey).get("bytecount");
       tempFeature2 = (ArrayList<Integer>) fea-
tures.get(fkey).get("packetcount");
       tempFeature3f = (ArrayList<Integer>) fea-
tures.get(fkey).get("IPDForward");
       tempFeature3b = (ArrayList<Integer>) fea-
tures.get(fkey).get("IPDBackward");
```

```
        sumipd = 0;
        System.out.println("\n\n" + strFlowInterval + "\t" + strValue);
        //System.exit(0);
        f3fLen = tempFeature3f.size();
        f3bLen = tempFeature3b.size();
        if ( f3fLen == 0 ) {
            ipdf = 0;
            varf = 0;
            //return;
        } else if (f3fLen == 1) {
            ipdf = tempFeature3f.get(0);
            varf = 0;
        } else {
            Collections.sort(tempFeature3f);
            for (int i = 0; i < f3fLen - 1; i++) {
                tempipd = tempFeature3f.get(i + 1) - tempFeature3f.get(i);
                tempFeature3f.set(i, tempipd);
                sumipd += tempipd;
            }
            ipdf = (double) sumipd / (double) (f3fLen - 1);
            sumipd = 0;
            for (int i = 0; i < f3fLen - 1; i++) {
                sumipd += Math.pow(ipdf - tempFeature3f.get(i), 2);
            }
            varf = Math.sqrt((sumipd / (double) (f3fLen -1)));
        }
        sumipd = 0;
        if ( f3bLen == 0 ) {
            ipdb = 0;
            varb = 0;
            return;
        } else if (f3bLen == 1) {
            ipdb = tempFeature3b.get(0);
            varb = 0;
        } else {
            Collections.sort(tempFeature3b);
            for (int i = 0; i < f3bLen - 1; i++) {
                tempipd = tempFeature3b.get(i + 1) - tempFeature3b.get(i);
                tempFeature3b.set(i, tempipd);
                sumipd += tempipd;
            }
            ipdb = (double) sumipd / (double) (f3bLen - 1);
            sumipd = 0;
            for (int i = 0; i < f3bLen - 1; i++) {
                sumipd += Math.pow(ipdb - tempFeature3b.get(i), 2);
            }
            varb = Math.sqrt((sumipd / (double) (f3bLen -1)));
        }
        int avBytesF = ((int)
Math.round((double)tempFeature.get(0)/(double)tempFeature2.get(0)));
        int avBytesB = ((int) Math.round((double) tempFea-
ture.get(1)/(double)tempFeature2.get(1)));
        String label = classify(IP1, IP2);
        if (label == "unknown") {
            continue;
        }
        // output value format :  bcF, pcF, avBytesF, bcB, pcB, avBytesB,
ipdF, ipdB, varF, varB, #optional diverse_subnets, dnsTotal_count, re-
set_count
        strValue =  tempFeature.get(0) + delimeter + tempFeature2.get(0) +
 delimeter + avBytesF + delimeter
                + tempFeature.get(1) + delimeter + tempFeature2.get(1) +
 delimeter + avBytesB +
                delimeter + (int) Math.round(ipdf) + delimeter + (int)
 Math.round(ipdb) + delimeter + (int) Math.round(varf) + delimeter + (int)
 Math.round(varb)
                + hostAttr + delimeter + label;
        //System.out.println("average bytes forward:\t" + avBytesF + "\t
 backward:\t" + avBytesB);
        //System.out.println(strValue);
        //System.exit(0);
        String[] keyParts = fkey.split(delimeter);
        String subKeyfeature = keyParts[2] + delimeter + keyParts[3];
 // user ports as features
        //output.collect(new Text(strFlowInterval), new Text(strValue));
        output.collect(new Text(subKeyfeature), new Text(strValue));
    }
  }
  private static String classify(byte[] IP1, byte[] IP2) {
      String strIP1 = CommonData.longTostrIp(Bytes.toLong(IP1));
      String strIP2 = CommonData.longTostrIp(Bytes.toLong(IP2));
      // malicious labeling
      Set storm = new HashSet();
      storm.add("66.154.80.101");
      storm.add("66.154.80.105");
      storm.add("66.154.80.111");
      storm.add("66.154.80.125");
      storm.add("66.154.83.107");
      storm.add("66.154.83.113");
      storm.add("66.154.83.138");
      storm.add("66.154.83.80");
      storm.add("66.154.87.39");
      storm.add("66.154.87.41");
      storm.add("66.154.87.57");
      storm.add("66.154.87.58");
      storm.add("66.154.87.61");
      if (storm.contains(strIP1) || storm.contains(strIP2)) {
          return "storm";
      }
      Set waledac = new HashSet();
      waledac.add("192.168.58.136");
      waledac.add("192.168.58.137");
      waledac.add("192.168.58.150");
      if (waledac.contains(strIP1) || waledac.contains(strIP2)) {
          return "waledac";
      }
      Set zeus = new HashSet();
      zeus.add("10.0.2.15");
      if (zeus.contains(strIP1) || zeus.contains(strIP2)) {
          return "zeus";
      }
      // non-malicious p2p labeling
      Set emuleNutorrent = new HashSet();
      emuleNutorrent.add("192.168.1.2");
      emuleNutorrent.add("192.168.3.2");
      if (emuleNutorrent.contains(strIP1) || emuleNutor-
 rent.contains(strIP2)) {
          return "emule";
          //return "utorrent";
      }
```

```
      Set vuzeNfrostwire = new HashSet();
      vuzeNfrostwire.add("192.168.2.2");
      vuzeNfrostwire.add("192.168.4.2");
      if (vuzeNfrostwire.contains(strIP1) || vuzeNfrost-
wire.contains(strIP2)) {
          return "vuze";
          //return "frostwire";
      }
      // IPs conflict with previous p2p apps
      Set skype = new HashSet();
      skype.add("192.168.0.4");
      skype.add("192.168.2.2");
      skype.add("192.168.1.2");
      skype.add("192.168.3.2");
      skype.add("192.168.4.2");
      skype.add("192.168.6.2");
      skype.add("192.168.5.2");
      skype.add("128.192.76.181");
      skype.add("128.192.76.182");
      skype.add("97.81.96.137");
      if (skype.contains(strIP1) || skype.contains(strIP2)) {
          return "skype";
      }
      return "unknown";
 }
private JobConf getFlowGenJobConf(String jobName, Path inFilePath, Path
Output){
    //Path Output = new Path(jobName);
    conf.setJobName(jobName);
    //conf.setNumMapTasks(16);
    conf.setNumReduceTasks(16);
    conf.setMapOutputKeyClass(BytesWritable.class);
    conf.setMapOutputValueClass(BytesWritable.class);
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(Text.class);
    conf.setInputFormat(PcapInputFormat.class);
    //conf.setOutputFormat(BinaryOutputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    conf.setMapperClass(Map_FlowGen.class);
    //conf.setCombinerClass(Reduce_FlowGen.class);
    conf.setReducerClass(Reduce_FlowGen.class);
    FileInputFormat.setInputPaths(conf, inFilePath);
    FileOutputFormat.setOutputPath(conf, Output);
    return conf;
}
public void startAnalysis (Path inputDir, Path outputDir,long cap_start,
long cap_end) throws IOException {
    FileSystem fs = FileSystem.get(conf);
    JobConf fGenJobconf = getFlowGenJobConf("PcapFlowStats", inputDir,
outputDir);
    fGenJobconf.setLong("pcap.file.captime.min", cap_start);
    fGenJobconf.setLong("pcap.file.captime.max", cap_end);
 // delete any output that might exist from a previous run of this job
    if (fs.exists(FileOutputFormat.getOutputPath(fGenJobconf))) {
      fs.delete(FileOutputFormat.getOutputPath(fGenJobconf), true);
    }
    JobClient.runJob(fGenJobconf);
    /*
    Path fGenOutputDir = FileOutputFormat.getOutputPath(fGenJobconf);
    System.out.println(fGenOutputDir.toString());
    JobConf fReduceJobConf = getFlowStatsJob-
Conf("PcapPeriodicFlowStats_red", fGenOutputDir, outputDir);
    // delete any output that might exist from a previous run of this job
    if (fs.exists(FileOutputFormat.getOutputPath(fReduceJobConf))) {
      fs.delete(FileOutputFormat.getOutputPath(fReduceJobConf), true);
    }
    JobClient.runJob(fReduceJobConf);
    */
}
}
```

PacketFlowStats.java:

```
import java.io.InputStream;
import java.net.URI;
import java.util.Calendar;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.JobConf;
import p3.common.lib.BinaryUtils;
import p3.common.lib.Bytes;
//import p3.ip.analyzer.P3CoralProgram;
public class PacketFlowStats {
static final String INPATH = "pcap_in";
static final String OUTPATH = "PcapFlowStats_out";
private static final int PCAP_FILE_HEADER_LENGTH = 24;
private static final int ONEDAYINSEC = 432000;
static JobConf conf = new JobConf(FlowAnalyzer.class);
public static void main(String[] args) throws Exception{
char argtype = 0;
String[] end = null;
long cap_start = Long.MAX_VALUE;
long cap_end = Long.MIN_VALUE;
String srcFilename = new String();
boolean rtag = false;
String dstFilename= OUTPATH+"/";
int windowSize = 600;
boolean fh_skip = true;
conf.addResource("p3-default.xml");
/* Argument Parsing */
int i = 0;
while(i<args.length){
    if(args[i].startsWith("-")){
        argtype = args[i].charAt(1);
        switch (argtype){
        case 'B': case 'b':
            String[] begin = args[i].substring(2).trim().split("-");
            if(begin.length<3)
                begin = args[i].substring(2).trim().split("/");
            if (begin.length == 3) {
                Calendar cal = Calendar.getInstance( );
                cal.set(Integer.parseInt(begin[0]),
                        Inte-
ger.parseInt(begin[1]),Integer.parseInt(begin[2]));
                cal.add(Calendar.MONTH, -1);
                cal.add(Calendar.DATE, -1);
                cap_start = Math.round(cal.getTimeInMillis()/1000);
            }
```

```
                break;
            case 'E': case 'e':
                end = args[i].substring(2).trim().split("-");
                if(end.length<3)
                    end = args[i].substring(2).trim().split("/");
                if (end.length == 3) {
                    Calendar cal = Calendar.getInstance( );
                    cal.set(Integer.parseInt(end[0]),
                            Inte-
ger.parseInt(end[1]),Integer.parseInt(end[2]));
                    cal.add(Calendar.MONTH, -1);
                    cal.add(Calendar.DATE, 1);
                    cap_end = Math.round(cal.getTimeInMillis()/1000);
                }
                break;
            case 'R': case 'r':
                srcFilename += args[i].substring(2);
                rtag = true;
                break;
            case 'D': case 'd':
                System.out.println(dstFilename);
                dstFilename = args[i].substring(2);
                System.out.println(dstFilename);
                break;
            case 'W': case 'w':
                windowSize = Integer.parseInt(args[i].substring(2).trim());
                conf.setInt("pcap.record.rate.windowSize", windowSize);
                break;
            default:
                break;
            }
        }
        i++;
}
if (srcFilename == null) srcFilename = INPATH + "/";
/* if there's input path*/
if(rtag){
    InputStream in = null;
    Path inputPath = new Path(srcFilename);
    FileSystem fs = FileSystem.get(URI.create(srcFilename), conf);
    byte[] buffer = new byte[PCAP_FILE_HEADER_LENGTH];
    long timestamp = 0;
    /* extract capture time */
    if(cap_start == Long.MAX_VALUE){
        FileStatus stat = fs.getFileStatus(inputPath);
        if(stat.isDir()){
            FileStatus[] stats = fs.listStatus(inputPath);
            for(FileStatus curfs : stats){
                if(!curfs.isDir()){
                    System.out.println(curfs.getPath());
                    in = fs.open(curfs.getPath());
                    if(fh_skip)
                        in.read(buffer, 0, PCAP_FILE_HEADER_LENGTH);
                    in.read(buffer, 0, 4);
                    timestamp =
Bytes.toInt(BinaryUtils.flipBO(buffer,4));
                    if(timestamp < cap_start)
                        cap_start = timestamp;
                    if(timestamp > cap_end)
                        cap_end = timestamp;
                }
            }
            in.close();
            fs.close();
            cap_end = cap_end + ONEDAYINSEC;
        } else {
            in = fs.open(inputPath);
            if(fh_skip)
                in.read(buffer, 0, PCAP_FILE_HEADER_LENGTH);
            in.read(buffer, 0, 4);
            timestamp = Bytes.toInt(BinaryUtils.flipBO(buffer,4));
            System.out.println(timestamp);
            cap_start = timestamp;
            if(cap_end == Long.MIN_VALUE){
                cap_end = cap_start+ONEDAYINSEC;
            }
            in.close();
            fs.close();
        }
    }
    if(cap_end == Long.MIN_VALUE)
        cap_end = cap_start+ONEDAYINSEC;
    Path outputDir = new Path(dstFilename);
    System.out.println(dstFilename);
    //System.exit(0);
    FlowAnalyzer fwAnalysis = new FlowAnalyzer(conf);
    fwAnalysis.startAnalysis(inputPath, outputDir, cap_start, cap_end);
}
}
}
```

Acknowledgement

First of all, I'd like express my deepest appreciation and gratitude to Professor Kwangjo Kim for his help, instructions, and support during the last two years. Moreover, I'd like to thank Professor Kim for letting me to work on topics of my interest and kindly guiding me through this process. Also I'd like to thank all the valuable members of Cryptology and Information Security lab for their warmest attitude, enthusiasm, and help during all this time. I enjoyed a lot working with each and every one of you.

Furthermore, I'd like extend my very appreciation to Professor Young Hee Lee and Professor Soontae Kim for their insightful comments and suggestions on my thesis work. Special thanks to Professor Kwangjo Kim for actively monitoring and helping to revise my thesis till the final day. I'm also greatly thankful to Professor Paul D. Yoo for his valuable reviews and suggestions on different Data Mining topics. Further, huge credit goes to Professor Mik Fanguy for invaluable contribution to my scientific writing skills.

Also I'm very grateful for all the acquaintances I made at KAIST and generally in Korea. Thank you very much to my roommates during my BS and MS degrees. I learned a lot and earned good friends by being your roommate. A great deal of credit goes to Azerbaijani community at KAIST. We've spent most of leisure time together exploring Korean culture, food, language, etc.

Last but not least, I'd to express my warmest gratitude to my family for their inspiration and support in all my endeavors. They always believed in me, and I experienced their candid acts of care at all times. Moreover, I'd like to express my gratefulness to all the international community of KAIST and generally to all the people I met during my stay in Korea. Definitely, my journey wouldn't be complete without any of you.

Curriculum Vitae

Name:	Khalid Huseynov
Date of Birth:	February 12, 1989
Birthplace:	Baku, Azerbaijan
Domicile:	70 Moscow avenue, Baku, Azerbaijan
Address:	KAIST 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea

Education

2008.09 – 2013.02	Department of Computer Science, KAIST (B.S), Republic of Korea
2013.03 – 2015.02	Department of Computer Science, KAIST (M.S), Republic of Korea

Career

2011.06 – 2011.08	Summer research internship program at Machine Vision Group, Seocho R&D campus, LG Electronics, Seoul, Republic of Korea.
2012.07 – 2012.09	IAESTE summer training program in Multimedia Lab, Ghent Univ. Ghent, Belgium.
2013.03 – 2015.02	Graduate Research Assistant in Cryptology and Information Security Lab under supervision of Professor Kwangjo Kim, KAIST.
2013.03 – 2013.06	Graduate Teaching Assistant in CC510 (Introduction to Computer Application) by Professor Soon Joo Hyun, leading C programming session for foreigners, Department of Computer Science, KAIST.
2013.09 – 2013.12	Graduate Teaching Assistant in CC500 (Scientific Writing) by

	Professor Mik Fanguy, KAIST.
2014.03 – 2014.06	Graduate Teaching Assistant in CC500 (Scientific Writing) by Professor Mik Fanguy, KAIST.
2014.09 – 2014.12	Graduate Teaching Assistant in CC500 (Scientific Writing) by Professor Mik Fanguy, KAIST.

Projects

2013.03 – 2013.10	Securing SCADA Protocols for Nuclear Plants
2013.05 – 2013.12	Intrusion Detection System for Critical Infrastructures
2014.01 – 2015.02	생체모방 알고리즘(Bio-inspired Algorithm)을 활용한 통신기술 연구
2014.07 – 2015.02	Intrusion Detection System for Critical Infrastructures Using Big Data Analytics

Publications

1. Khalid Huseynov, Kwangjo Kim, and Paul D. Yoo, “Evaluation of Public Datasets for IDS/IPS Benchmark”, Proc. of KIISC (Korean Institute of Information Security) Conference, SunChunHyang Univ., CheonAn, Korea, Sep. 27, 2013
2. Khalid Huseynov, Kwangjo Kim, and Paul D. Yoo, "Semi-supervised Botnet Detection Using Ant Colony Clustering", 2014 Symposium on Cryptography and Information Security (SCIS 2014), Jan. 21-24, 2014, Kagoshima, Japan.
3. Khalid Huseynov, Kwangjo Kim,"Unsupervised Hadoop-based P2P Botnet Detection with Threshold Setting", 한국정보보호학회 동계학술대회(CISC-W'14), 2014.12.06. 한양대학교, 서울. - [우수논문]