UNIVERSITY OF NAIROBI

College of Biological and Physical Science

School of Computing and Informatics

Near Real Time Machine Driven Signature Detection, Generation and Collection

By

EDWIN OUMA NGWAWE

REG.NO. P53/65229/2013

SUPERVISOR: DR. ELISHA A. ABADE

Project documentation submitted in partial fulfillment of the requirements for the award of a degree in Master of Science in Distributed Computing Technology

STUDENT DECLARATION

This research project is my original work and has not been presented for award of any degree in any University

……. ……

Signature Date
Edwin Ouma Ngwawe

This research project has been submitted for examination with my approval as University Supervisor

…… …….. ……..……

Signature Date
Dr. Elisha Abade.

ACKNOWLEDGEMENT

I wish to acknowledge the Almighty God for strength, safety and guidance throughout this course.

I also wish to acknowledge my Supervisor Dr Elisha O. Abade for his critic, time, patience, direction and guidance during the course of the project.

I also acknowledge my colleagues and friends at my place of work for their understanding during my absence from work as I worked on this project.

Acknowledge my family for their understanding, patience and support.

Finally I wish to acknowledge my fellow students at School of Computing and Informatics and presentation panelists for their criticisms, correction and suggestions on this project.

ABSTRACT

Internet worms can spread very fast and cause losses both in terms of lost business opportunities as well as human resources required to alleviate the caused damages. There exists two ways of protecting against the worms namely anomaly based and signature based systems. Signature based systems depends on security signatures (patterns) that match particular known attacks while anomaly based systems relies on detecting anomalies with the background idea that abnormal activity is malicious. With the ever increasing internet speeds and growing complexity of data across it, it is necessary to have correspondingly fast ways of analyzing network traffic in order to categorize activities in time. Also the existence of zero-day attacks makes relying of preconfigured signatures unreliable. This study sought to find how to develop an accurate, robust near real time machine driven Internet security signature detection, generation and collection system using big data technologies such as Hadoop Map Reduce programming model and Hadoop Distributed File System. We set up Hadoop Ecosystem at the University of Nairobi Laboratory and gathered and analyzed both malicious and innocuous network traffic and generated documented security signatures for known Internet worms with near real time speeds and also corresponding signatures for synthetic worms to simulate zero-day worms. We realize that adding the number of nodes to the Hadoop cluster not only increases the processing speeds but also eases the resources for the signature generation system. The increased power of the system improves accuracy and the HDFS replication improves system robustness.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

COV - Worm Coverage of the malicious flow

COVmin – minimum desirable worm coverage

FP – False positive ratio

FPmax – maximum acceptable degree of false positives

M – Malicious flow dataset

N – Normal flow dataset

Definition of Terms

Real-time

The actual time during which a process or event occurs.

Near Real-time

Pertaining to the timeliness of data or information which has been delayed by the time required for electronic communication and automaticdata processing. This implies that there are no signifi cant delays.

Internet Worms

An internet worm is a program that spreads across the internet by replicating itself on computers via their network connections.

Worm signature

Worm signatures are the binary pattern of the machine code of a particular worm.

Worm Signature Detection

Worm signature detection is looking for patterns in network traffic and using them to decide whether or not the traffic contains internet worms.

Worm Signature Generation

Worm signature generation is the process of successfully producing signatures that match worms.

Worm Signature Collection

Worm signature collection is the process of picking generated worm signatures and deploying them to security appliances such as firewalls and the intrusion detection systems.

CHAPTER ONE: INTRODUCTION

A worm program is an independently replicating and autonomous infection agent, capable of seeking out new hosts and infecting them via the network. (Nazario, 2004). Because of evermore pervasive Internet connections and software monoculture, worms with their typical scan/compromise/replicate pattern can infect all the vulnerable hosts in a matter of few hours or even minutes (Staniford et al., 2002). To contrast such a threat, the research community has proposed different kind of Intrusion Detection Systems (IDSs) (Axelson, 2000). For example, a misuse-based IDS, deployed at the gateway between its network and the Internet, may filter incoming and outgoing network traffic for known signatures that correspond to malicious flows samples (Roesch, 1999), (Paxson, 1998). In the past, signatures used by IDS have been generated manually with the supervision of security experts which studied network traces and inferred worm's signatures. However, in the last few years, the frequency and virulence of worm's outbreaks increased dramatically thanks to their improved efficiency and evasion methods, and became well-understood that signatures generation processes that involve human labor were not feasible (Zou et al., 2003) anymore.

To face the slow pace of this approach in signatures generation and to speed up this task, automatic signature generation systems have been proposed in the past years. Systems like Honeycomb (Kreibich & Crowcroft, 2003), Autograph (Kim & Karp, 2004), EarlyBird (Singh et al., 2004), Polygraph (Newsome et al., 2005), Hamsa (Li et al., 2006) and Lisabeth (Cavallaro et al., 2008) can monitor network traffic to identify new Internet worms and produce signatures for them. Such systems use content based analysis to generate signatures. The increasing speeds of the internet as per IEEE report (IEEE, 2014) require that signature generation systems be able to keep pace with high speeds in order to generate the signatures in real-time.

This study seeks to develop an Internet worm signature generator that will work with near real-time speeds considering the ever growing amount of data across the internet.

1.1 Problem Statement.

One of the reasons that make securing of information systems such a big (and sometimes frustrating) challenge is the huge amount of data that has to be analyzed in order to protect the systems accurately, these data come in the forms of network traffic, application logs and even network event logs. The growing amount of data to be analyzed would normally introduce delays in the analysis and hence classification of activities in the distributed information system. Such delays unless taken care of can lead to late reporting or detection which may give enough time for the malicious activity to cause catastrophic effects to the system.

Another challenge is the growing complexity of information system that makes it too difficult for an administrator to observe and notice anomalies manually hence the requirement for a machine assisted way of analyzing the security data.

1.2 Purpose

The purpose of this study is to come up with a system which can automatically generate signatures, detect malicious activities based on these signatures and take a corrective measure with near real time speeds.

1.3 Research objectives

The objectives of this research are:

(1) Find how to develop a system that can generate security signatures in distributed systems with near real time speeds considering the contemporary large amounts of data.
(2) To develop a prototype for the system described in part one above.
(3) To evaluate the performance of our prototype.

1.4 Research Questions

Every objective of this research is achieved by answering a set of questions related to as below:

The questions are:

1) How can we automatically generate distributed systems security signatures?
2) How can we attain near real-time processing speeds in distributed systems?
3) How can we attain an accurate security signature generator in distributed systems?

4) How can we attain a robust security signature generation systems?
5) How can we present signatures for collection after generation?

1.5 Scope

This study concentrated on automatic signature generation, detection and collection for Internet worms using big data technologies.

1.6 Significance of the Study

The study is expected to provide a prototype for a product which can be used to generate network security signature using big data technologies, considering the contemporary large amounts of data across the internet and provide the signatures for deployment and thus enhance security of contemporary information systems.

The study also aims to provide a prototype to a system that will reduces the human efforts hence human resource requirements that are needed to monitor the information system effectively. This translates to savings in running costs hence long term savings.

CHAPTER: TWO LITERATURE REVIEW

2.1 Overview of Intrusion Detection Systems

According to Wikipedia, (Wikipedia Encyclopedia, 2014) an intrusion detection system (IDS) is a device or software application that monitors network or system activities for malicious activities or policy violations and produces reports to a management station. IDS come in a variety of "flavors" and approach the goal of detecting suspicious traffic in different ways. There are network based (NIDS) and host based (HIDS) intrusion detection systems. Some systems may attempt to stop an intrusion attempt but this is neither required nor expected of a monitoring system. Intrusion detection and prevention systems (IDPS) are primarily focused on identifying possible incidents, logging information about them, and reporting attempts. In addition, organizations use IDPSes for other purposes, such as identifying problems with security policies, documenting existing threats and deterring individuals from violating security policies. IDPSes have become a necessary addition to the security infrastructure of nearly every organization.

IDPSes typically record information related to observed events and notify security administrators of important observed events and produce reports. Many IDPSes can also respond to a detected threat by attempting to prevent it from succeeding. They use several response techniques, which involve the IDPS stopping the attack itself, changing the security environment (e.g. reconfiguring a firewall) or changing the attack's content.

IDS can either use Anomaly-based Detection or Signature-based (Misuse) Detection methods to classify the traffic.

2.1.2 Differences between signature based and anomaly based IDS

According to Boriana Ditcheva and Lisa Fowler (Ditcheva & Fowler, 2005) the below illustrates the differences in how signature based and anomaly based IDSes work.

Signature Based IDS	Anomaly based IDS
• Looks for specific and explicit indications of attacks – Identified by raw byte sequences (strings), protocol type, port numbers, etc. • Low false positives	Central idea: "abnormal" = "suspicious" • Automatically learns • Detects novel attacks (and its variations) • Can be left to run unattended • Requires a notion and definition of "normal"

– "Knows for a fact" what is suspicious, what is normal • Detects only behavior that was previously defined to be suspicious – Can have tight signatures (high confidence) • Simple and efficient process • Easy to share – Repositories of signatures	• Susceptible to false negatives – Unusual is not necessarily illicit/malicious – Usual is not necessarily benign • e.g. attacks that manifest slowly • Computation intensive

2.1.3 IDS Placement

2.1.3.1 Outside firewall

– Detects all attacks directed at your network.

– Detects more events.

– Generates more logs.

2.1.3.2 Inside firewall

– Only detects what the firewall lets in.

– Less state information.

2.1.4 Sample IDS Architecture

Figure 1 below shows sample architecture, a case of Snort IDS.

Here, incoming network traffic is passed across preconfigured rules and if they match the rules, a predetermined action is taken; else the traffic packets are dropped. The action may be to forward a packet to a particular subsystem for processing or further analysis.

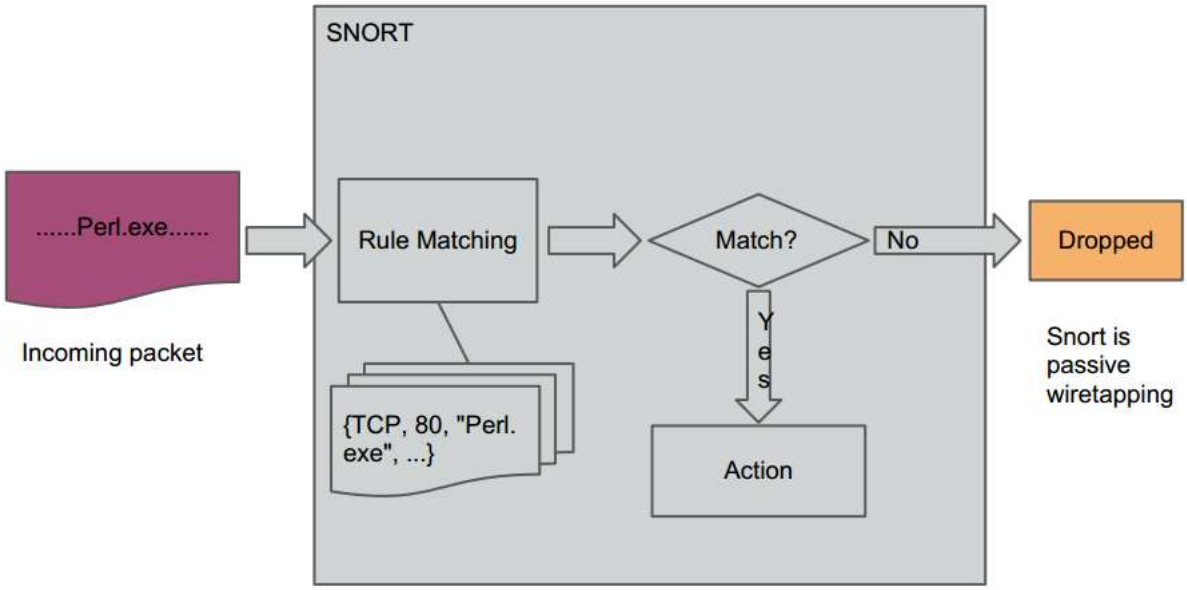


Figure 1 Sample IDS architecture, case of Snort IDS

2.1.5 Overview of a computer worm

According to Wikipedia (Wikipedia Inc, 2013), a computer worm is a standalone malware computer program that replicates itself in order to spread to other computers. Often, it uses a computer network to spread itself, relying on security failures on the target computer to access it. Unlike a computer virus, it does not need to attach itself to an existing program. Worms almost always cause at least some harm to the network, even if only by consuming bandwidth, whereas viruses almost always corrupt or modify files on a targeted computer.

Containment of worm outbreaks need to be automated, it should not be left such that only after a new worm is first detected and analyzed by a human who identifies a signature can the containment process be initiated. Zhou Sing in their study, (Singh et al., 2003) indicates slow response and constant human efforts as the challenges with human mediated signature generation, detection and collection, in fact Moore in there study, (Moore et al., 2003) show that detection and containment must be initiated within minutes or seconds to prevent wide-spread infection in a 24 hour period.

Singh (Singh et al., 2003) continues to list the following as the ideal design goals for worm detection and containment system.

(i) Real time detection: Detection of a worm should be within seconds and potentially initiate some containment measure within similar time frame. This prevents widespread infection before it is too late.

(ii) Content Agnosticism: The detection system should not rely on external manually supplied worm signature, but instead automatically extract the new worm signature for the worms that may occur in future. This is necessary to reduce constant human and other overheads involved in detecting the worm.

(iii) Versatile Deployment: The generic model should be deployable (with some modification) at any point in the network including routers (edge and core), intrusion detection systems running on local area networks and even systems themselves. This is necessary to enable effective action against spreading of a worm.

(iv) Scalability: To support versatile deployment, worm detection and containment must be implementable in high speed devices such as the core routers. Ideally worm detection and containment mechanisms should be scalable, it should use moderate amount of memory and should not depend on per-flow state. Moreover, any packet processing should sufficiently small and parallelizable to low-line rate implementations at 1 and 10 Gbps.

(v) Resilient to simple worm changes: Much as we do not expect a perfect system, simple countermeasures taken by worm authors should be taken care of by the system.

(vi) Ability to Handle Asymmetric Routing: Should be able to work even if they can see only one direction of flow, e.g. in core routers.

2.1.6 Sample signatures for known worms and synthetic worms

The following are sample security signatures for some known Internet worms.

Worm name	Signature
Code-Red II	{'.ida?': 1, '%u780': 1, ' HTTP/1.0\r\n': 1, 'GET /': 1, '%u': 2}
Apache-Knacker	{'\xff\xbf': 1, 'GET ': 1, ': ': 4, '\r\n': 5, ' HTTP/1.1\r\n': 1, '\r\nHost: ': 2}
ATPhttpd	{'\x9e\xf8': 1, ' HTTP/1.1\r\n': 1, 'GET /': 1}

Table 1 Sample security signature for known real internet worms

Since the study is interested in zero-day (unknown) polymorphic worms, it would be interesting to include some synthetic worms, which would be used to simulate the currently unknown worms.

Polymorphic engines can be used on the source codes of known worms to achieve their polymorphic versions (Li et al., 2006).

The table below shows synthetic worm signature samples

CLET	{ '0\x8b': 1, '\xff\xff\xff': 1, 't\x07\xeb': 1}
TAPiON	{'\x00\x00': 1, '\x9b\xdb\xe3': 1}

Table 2 Sample security signature for known synthetic internet worms

2.2 Worm Signature Generation

The goal is to find a pattern or signature that can allow for the detection of a specific attack, just the way computer virus detection systems works.

The generated signature has to be sensitive in order to reduce false negatives and specific enough to cover as many of the variants as possible while minimizing false positives.

Many approaches have been used to automatically generate worm signatures. Categorically, content based and behavior based approaches have been used as shown in (Li et al., 2006).

Content based approaches involve monitoring the patterns of the content or the payload being passed across a particular point of inspection. Occurrence of a particular content in more than a specified threshold number of times in a given set of traffic flows may be used as criteria to mark it as either suspicious or innocuous. Behavior based systems involve observing the behavior of data traffic in a particular system, for instance if the particular packet attempts to exploit know vulnerabilities such as buffer overrun or attempting to overwrite a return address of a particular function, then the particular set of traffic can be classified as either suspicious or innocuous. Activities such as when a system is sending so much port scan requests can also constitute a criterion for classification of the data as either suspicious or innocuous.

Techniques of deep packet inspection, also called Information eXtraction or IX are used to achieve traffic classification. Here the both the data or the payload part of the packet as well as its header is inspected as it passes though a particular point of inspection hence making its classification possible. One may also use stateful packet inspection engines or firewalls to classify packets based on the behavior, for example when some so much of a given set of packets are attempting to go through from one host to another when there is no TCP session between the two hosts.

Tools have been developed to achieve packets reassembly into network flows and classify the same flows as either malicious or innocuous. Technical report written by (Bujlow et al., 2013) provides a comparison of deep packet inspection tools for classification hence one may use this as a guideline for choosing a traffic classifier. They found PACE as the best network traffic classification solution among all the tools studied. Among noncommercial tools they chose Libprotoident or NDPI depending on the scenario studied.

The classified traffic may then be used to generate signatures based on a variety of algorithms as discussed below.

Behavior based mechanisms for signature generation include the (Newsome & Song, 2005) which marks data from un-trusted sources such as networks and emulates the behavior of the application in an application emulator such as in Valgrind, later the system observes whether the marked data is used to exploit security vulnerabilities such as format string exploits or buffer overrun and then signatures are generated from the behaviors of the offending data. These are

majorly host based detection systems which require very high computational efforts since the system needs to emulate the entire behavior of the application it is monitoring

2.2.1 Polymorphic Worms

When using content based signature detection techniques, it is important to consider that some worm authors deliberately vary the worm contents (polymorphic worms) and therefore take care of this phenomenon.

2.2.2 Early Signature Generation Systems

Early systems such as the EarlyBird (Singh et al., 2003) HoneyComb (Kreibich & Crowcroft, 2003) and Autograph (Kim & Karp, 2004) do not capture polymorphic worms well since they are based on the assumption that the worm signature will be based on some long common invariant string which is not usually the case.

2.2.3 Polygraph Signature Generation System

Polygraph (Newsome et al., 2005) captures polymorphic worms with reasonable speeds and accuracy. It is based on the fact that for successful attacks to take place, some parts of the payload will ultimately be invariant. Here the payload is categorized into the invariant bytes which have fixed values and when changed, the worm will not succeed in achieving its goal, wildcard bytes which are bytes that can take any form hence can be varied by the worm author as he desires, and the code bytes which are the polymorphic codes that is executed by the worm which are the output of the polymorphic code engine. Here the invariant bytes are used to generate worm signatures. Research by (Li et al., 2006) indicates that the polygraph can be enhanced to perform better in terms of speed and accuracy. They built Hamsa system to this effect.

2.2.4 Hamsa Signature Generation System

Hamsa is based on the assumption that a worm must exploit one or more server specific vulnerabilities. This constrains the worm author to include some *invariant* bytes that are crucial for exploiting the vulnerabilities. Hamsa authors formally capture this idea by means of an adversary model T which allows the worm author to include any byte strings for the worm flows as long as each flow contains tokens present in the invariant set I in any arbitrary order. Under certain uniqueness assumptions on the tokens in I the authors can analytically guarantee

signatures with bounded false positives and false negatives. Since the model allows the worm author to choose any bytes whatsoever for the variant part of the worm, Hamsa is provably robust to any polymorphism attack. Such analytical guarantees are especially critical when designing algorithms against a human adversary who is expected to adapt his attacks to evade our system. According to Hamsa authors, they were the first to provide such analytical guarantees for polymorphic worm signature generation systems.

To give a concrete example, Hamsa authors designed an attack which could be readily employed by an attacker to evade the then state-of-the-art techniques like Polygraph while Hamsa successfully found the correct signature.

The signature generation is achieved by simple greedy algorithms driven by appropriately chosen values for the model parameters that capture authors' uniqueness assumptions and are fast in practice. Compared with Polygraph, Hamsa is tens or even hundreds of times faster, as verified both analytically and experimentally. The C++ implementation can generate signatures for a suspicious pool of 500 samples of a single worm with 20% noise and a 100MB normal pool within 6 seconds with 500MB of memory. Thus Hamsa can respond to worm attacks in its crucial early stage. Hamsa authors also provided techniques for a variety of memory and speed trade-offs to further improve the memory requirements. For instance, using MMAP we can reduce the memory usage for the same setup to about 112MB and increase the runtime only by around 5-10 seconds which is the time required to page fault 100MB from disk to memory. All the experiments were executed on a 3.2GHz Pentium IV machine.

In the absence of noise, the problem of generating conjunction signatures, as discussed by Polygraph, is easily solved in polynomial time. Presence of noise drastically affects the computational complexity. It is shown that finding multi-set signatures, which are similar to Polygraph's conjunction signatures, in the presence of noise is NP-Hard.

In terms of noise tolerance, can bound the false positive by a small constant while the bound on false negative depends on the noise in the suspicious traffic pool. The more accurate is the worm flow classifier in distinguishing worm flows from the normal flows; the better is the bound on false negatives that we achieve. A generalization for measuring the goodness of signature using

any reasonable scoring function was provided and this extends the analytical guarantees to that case.

Hamsa authors validated their model of worm flows experimentally and also proposed values for parameters characterizing the uniqueness condition using their experimental results. Evaluations on a range of polymorphic worms and polymorphic engines demonstrate that Hamsa is highly efficient, accurate, and attack resilient, thereby significantly outperforming Polygraph.

The figure 1 shows how to attach Hamsa Monitoring System to a high speed Router

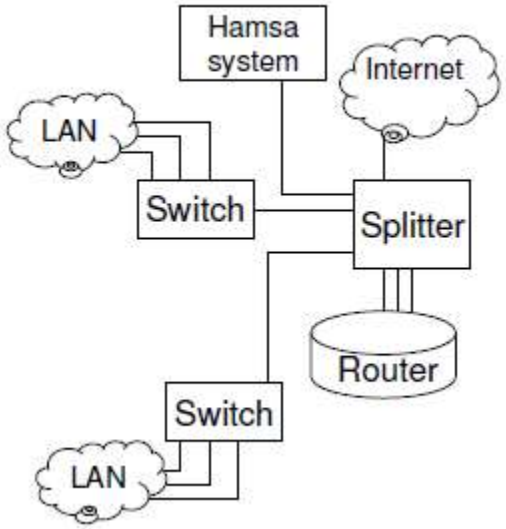


Figure 2 Attaching Hamsa signature generator system to high speed network

The system can fetch data through port span or via optical fiber split.

Traffic is then classified and used to generate signatures based on the algorithm.

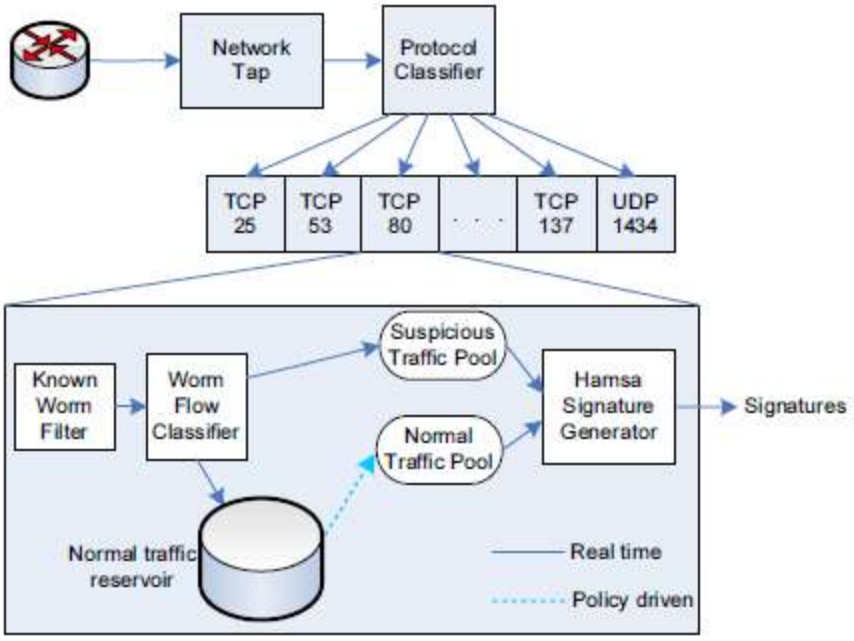


Figure 3 Worm Traffic Classification.

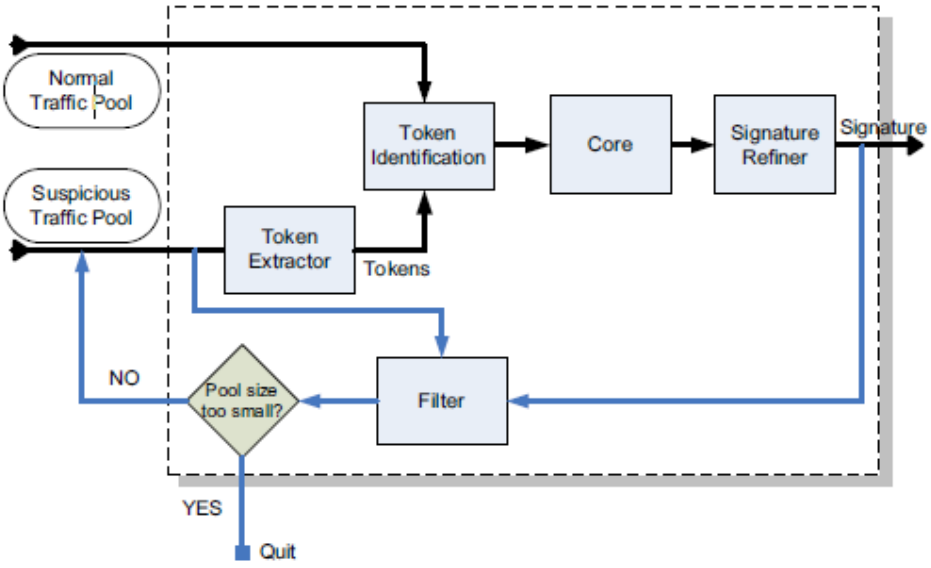


Figure 4 Hamsa signature generation module

2.2.5 Lisabeth Signature Generation System

The Hamsa system can be enhanced further into Lisabeth (Cavallaro et al., 2008) system by adding a dispersion filter modifying the signature generation module to the initial Hamsa architecture. The dispersion filter helps mitigate poisoning attacks, a situation where by a worm author deliberately inserts noise into the suspicious flow so as to mislead signature generator into generating signatures that actually belong to no real worm there by leaving the actual worms signatures since the generation algorithm relies on frequency of a particular content to prioritize its generation as a signature so if noise supersedes actual worm flow then the noise will be generated as a signature instead.

Figures 2.4 and 2.5 below show a high level architecture of Lisabeth system and the Lisabeth signature generator module respectively. The differences with Hamsa are marked in white.

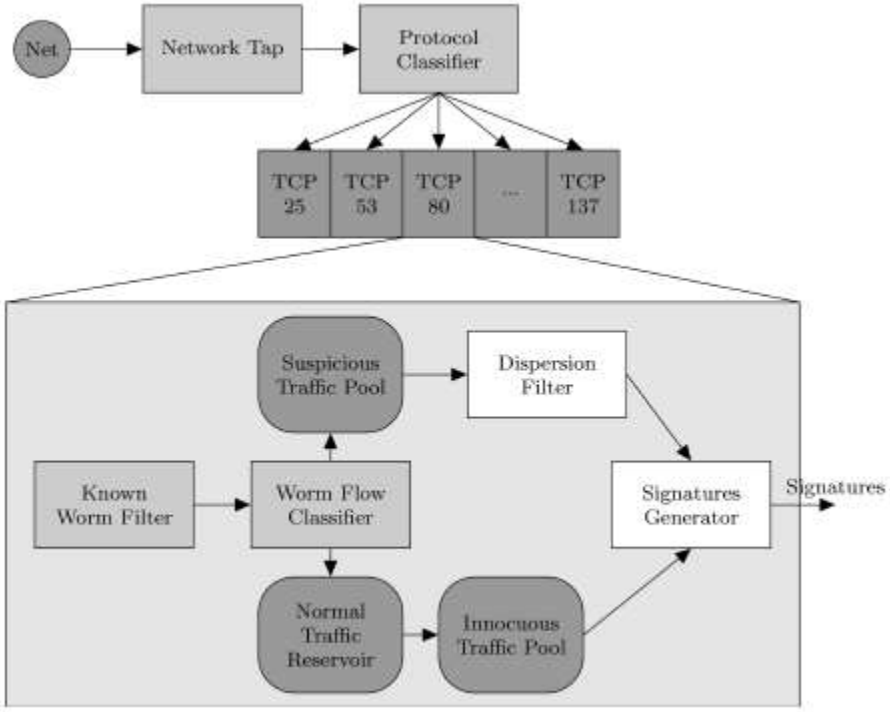


Figure 5 High level architecture for Lisabeth system.

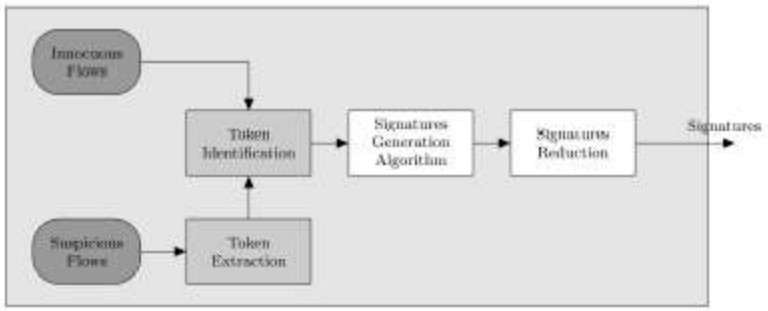


Figure 6 The Lisabeth signature generator module

The goal of this filter is to perform a dispersion analysis on suspicious flows in order to send to the generator a well dispersed set of flows.

By using this dispersion technique only fewer flows for each source address are sent to the signature generator. Thus, to be successful, worm instances that want to perform suspicious pool poisoning attacks against Lisabeth system must synchronize themselves with respect to the features to use in flows. This, as a consequence, makes the attack harder to carry out.

The signature generator module hosts the generation algorithm.

As it is possible to see from Figure 2.5, the first operation performed on the suspicious flows pool is token extraction. In this phase we find all sequences of at least length L bytes that occur in more than λ fraction of the suspicious pool. The constraint on sequences length is required to ignore too small tokens, while λ is used to take into account only those that occur in several flows.

To speed up the algorithm's execution time, all the extracted tokens are then identified in each innocuous flow, and all flows, innocuous and suspicious, are converted in sequences of tokens discarding all bytes sequences not included in the extracted tokens set. Flows so obtained are then sent to the signature generation algorithm that, as we will see in below, creates the required signatures.

The last phase performs signatures reduction on returned signatures to remove all tokens that, always appearing as sub tokens of others ones, are not required to enhance signatures specificity.

2.2.5.1 Lisabeth Signature Generation Algorithm

Given suspicious (M) and innocuous (N) flows pools, the aim of the signature generation algorithm is to find a set S of signatures S_i such that $FP_{si} \leq FP_{max}$ and $COV_{si} \geq COV_{min}$. To this end Hamsa adopts a purely greedy approach. However, building the most specific signature including all possible invariants giving a good coverage of M without having any knowledge of the nature, real or fake, of the invariants or using a greedy algorithm with a restricted view of global situation without having a global overview of all possible signatures may weaken the strength of the detection. To avoid generation of redundant signatures, we enforce an additional constraint in our algorithm:

$$s_i \in \mathcal{S} \Leftrightarrow \nexists s_j \in \mathcal{S} \mid \mathcal{M}_{s_i} \subseteq \mathcal{M}_{s_j}$$

The idea is to create all the signatures that match a considerable fraction of the suspicious pool, while avoiding the addition of new tokens to a partial signature when it has already an acceptable false positive rate (i.e., a rate less than FP_{max}). In this way, we can have more signatures per worm but we have surely at least one specific enough signature containing only a subset of I. The value used for FP_{max} may be smaller than the value used in Hamsa for the shortest signatures, i.e., composed by only one token, and so the maximum false positives rate, accepted for a single signature, is lower.

The generation of all the possible subsets of the extracted tokens and subsequent check of the given constraints would require a non-negligible computational effort, so another aim of the proposed algorithm is to avoid, whenever possible, to generate redundant or useless long signatures. As we can easily think, for any given token there may be more occurrences of it in a single flow. To avoid worrying about this problem, we assign to each pair (token, number of occurrence) a unique identifier. In this way we consider these occurrences as different tokens. It is important to note that the same occurrence of the same token in different flows will have the same identifier. All these matching information between identifiers and related (token, number of occurrence) pairs are stored in an appropriate data structure, called TM, for subsequent use.

Once this identifier is assigned, for each of the aforementioned pair, we build a list of all suspicious flows in which it occurs, create a partial signature with that pair and related flows list and insert this new partial signature into the partial signatures set PS.

To avoid generation of redundant signatures caused by the high number of tokens that appear always as subtokens of other ones, we remove these sub tokens from PS and take them into account at the end of the signature generation algorithm. We say that t_1 is a subtoken of t_2 if $t_1 \neq t_2$, t_1 is substring of t_2 and the occurrence number considered for t_1 is the same as of that considered for t_2. Then, if a token t_1 occurs as a subtoken of t_2 we delete partial signature containing t_1 and add t_1 in the subtokens list of t_2 stored in subtokens data structure ST.

Once the partial signature set is build, our algorithm proceeds iteratively. First it evaluates the false positives rate of all the available partial signatures. Those with a value low enough are inserted into S and deleted from PS. The remaining signatures are then merged into each other and if each new partial signature has not a good coverage of M, it is discarded along with those of the prior iteration. Coverage evaluation is simple and fast: the number of covered flows is given by the intersection between flows lists of the two partial signatures merged together. The merging of two partial signatures is performed only if the new one has just one more token than the two from which it is obtained.

Finally, iterations are stopped when the partial signature set is empty. Before returning it, to each generated signature are added the ignored subtokens of each included supertoken.

2.2.5.2 Lisabeth Signature Generator Attack Analysis

As discussed by Cavallaro (Cavallaro et al., 2008), although signature generation system such as Hamsa (Li et al., 2006) and Polygraph (Newsome et al., 2005) are also cable of generating good signatures including in the presence of noise, they do not behave very accurate if analyzed flows are provided by an attacker in an attempt to mislead the generation of signature with forged invariants.

There are three ways an attacker can use to mislead signature generation, namely target feature manipulation, suspicious pool poisoning, innocuous pool poisoning.

Target feature manipulation is the process by which an attacker seeks to manipulate some characterizing features of the worm, such as the worm code or the protocol framework bytes. There exist many techniques that can be used to minimize or obfuscate required features or to include additional spurious features into the worm samples to mislead signature generation.

Suspicious pool poisoning is the process whereby the adversary places some non-worm samples inside the suspicious pool. These samples are specially designed to mislead the signature generator.

Innocuous pool poisoning on the other hand is the process where an adversary paces specially crafted samples inside the innocuous pool to mislead the signature generation.

Systems like Hamsa and Polygraph suffer of some of these attacks. Since Hamsa is an improvement of Polygraph, we shall focus our discussion on Hamsa.

The greedy approach used in Hamsa's signature generation algorithm, with its incremental generation of partial signature can lead to building useless signatures, thus making these signatures unable to match any real worm sample. In fact the greedy algorithm proceeds iteratively by selecting at every iteration the token that when added to the previous ones gives the best signature, obtaining each of them by adding a token to the signature generated in the previous iteration.

The first signature contains only the token that maximizes COV rate within those offering an FP less than the given FPmax rate for the signature of that length. At each iteration, Hamsa's generation algorithm adds to the previous selected signatures the token with FP rate less than a threshold with them maximum COV rate. When the maximum length for a signature is reached, Hamsa selects the best one by evaluating a score for each signature. This score takes into account FP and COV rate and signature's length. The one with the higher score is then selected and returned as signature for the given input. This approach presents some weaknesses.

Let us denote W as a worm and

$$\mathcal{I} = \{I_a, I_b, I_c, \ldots, I_x\}$$

as a set of its invariants. An attacker can try to introduce some fake invariants

$$\mathcal{F} = \{F_1, F_2, F_3, \ldots, F_y\}$$

I.e. tokens found in the suspicious flow but not rally required to for an exploit to succeed. In order to ensure that Hamsa considers only fake invariants (and neglecting real ones), the elements of F can be forged to the following constraints.

$$FP_{\{F_1,F_2,F_3,\ldots,F_i\}} \leq u(i) \quad \forall i \in [1..y]$$

where u(i) is the function Hamsa uses to determine if the false positive rate of a given signature is low enough. The above constraint can be trivially respected. Given I and F, an instance of W will generate two classes of samples:

(i) Worm samples

$$W_1^{IF}, W_2^{IF}, \ldots, W_n^{IF}$$

That contains that contain real and fake invariants.

(ii) Non worm samples

$$W_1^{F}, W_2^{F}, \ldots, W_j^{F}$$

That contains only fake invariants and so is not real working worms.

To assure attack achievement, an attacker must send worm and non-worm samples to the victim and drives the initial classifier to classify these as suspicious. To the attacker will need to hold the anomalous traffic behavior depending on the classification methods used.

In case W sends n samples of W^{IF} and j samples of W^{F} within $n + j \geq \lambda$, where λ is the minimum number of token occurrences in suspicious flow pools required to be considered for signature generation, the token extraction procedure will extract all fake variants tokens F_I and also I_I if $n \geq \lambda$, all real invariants.

The signature generator will first choose a fake invariant since there is always at least on fake invariant in the sample i.e. F_I with a false positive rate less that u(i) that occurs more than any real invariant I_I.

Similarly, in the subsequent iterations, the algorithm will include in the temporary signature fake invariant because there is always an F_I that when added to the previous one will respect u(i) value and with the others, occur more times than any other real invariant.

To predict the way algorithm chose tokens and hence prevent it from picking the right token, the adversary needs to include non-worm samples y in the flow such that:

W_I^{Ford} contains only $F_{[1\ldots i]}$ such that fake invariants, i is a member of the set [1…y].

In this way, if n is greater than the maximum length of a signature considered by the algorithm (Hamsa proposes a length of 15 tokens), it is possible to obtain a signature made only by fake invariants.

The execution of the signature refinement procedure, that includes in the selected signature all the tokens occurring in all the covered suspicious flows only if not already present in the signature, does not affect attack effectiveness; at most all remaining fake invariants, and only these, will be included in the selected signature. This attack leads signature generation algorithm to build a signature that does not include any real invariant. The attacker is now able to send another burst of worm and non-worm samples without being detected. Even if more than one worm instance attacks the same host, this attack, unlike the well-known red herring attack (Newsome et al., 2006), can work any way if the value of n is big enough with respect to the value of j. In the red herring attacks, the adversary incorporates fake invariants into the worm samples to lead the generation system to create signatures that include those spurious features in addition to the necessary invariant tokens. Then the adversary can evade the resulting signature by not including some fake invariants in subsequently generated worm samples. So, if two or more not synchronized attackers send worm samples using different fake invariants sets, the signatures generation system will be able to create the correct signature. This is possible because the number of real invariants is greater in comparison to that of fake invariants and so, by offering better COV rate, they will be selected before the fake ones. The new signature probably contains only real invariants and so matches with all current and future flows.

2.2.5.3 Attack Effectiveness

Cavallaro, (Cavallaro et al., 2008) evaluated both Hamsa and the Lisabeth for this new attack by injecting 20 fixed different tokens to the variant part of each worm and non-worm sample.

Hamsa generated one signature built only with injected fake invariants. Due to the lack of real invariants presence in the signature, no new polymorphic instances of the same worm could be detected (100% false negative). In addition, all analyzed worm flows are then discarded, and so the system is not able to build a correct signature also in subsequent execution of the signature generation algorithm. This is not the case with Lisabeth model which we adopted.

2.2.4.4 Evaluation of Lisabeth Signature Generation System

Lisabeth evaluated for effectiveness and efficiency under several scenarios. To evaluate the effectiveness of our approach, a case where the suspicious flow pool contained only flows of one worm was first considered. Next, a case where suspicious flows pool contained some noise and some innocuous flows as well was considered. Finally, a case where the suspicious pool contained flows from multiple worms was considered. To evaluate Lisabeth efficiency the system was ran with different amounts of data both for suspicious and innocuous pools and compared these results with those of our Hamsa implementation. To accomplish the tests, polymorphic versions of three real-world exploits, i.e., the Apache-Knacker, the ATPhttpd, and the Code-Red exploit were used, generating suspicious flows using a companion tool included in Polygraph's source code. As innocuous flows HTTP traces collected from their laboratory network gateway during normal usage was used as well as HTTP traffic from web crawlers. During the evaluation process, several network flows pools of different sizes were used as input both for the suspicious and the innocuous pools.

Lisabeth was found to outperform Hamsa in both speed and accuracy.

2.3 Near Real Time Speeds

2.3.1 Need for Near Real Time Speeds

Center for Applied Internet Data Analysis, (Center for Applied Internet Data Analysis (CAIDA), 2003) reports on their website that the Sapphire Worm was the fastest computer worm in history, as it began spreading throughout the Internet, it doubled in size every 8.5 seconds and infected more than 90 percent of vulnerable hosts within 10 minutes. The worm infected at least 75,000 hosts, perhaps considerably more, and caused network outages and unforeseen consequences as such canceled airline flights, interference with elections, and ATM failures.

The worm achieved its full scanning rate (over 55 million scans per second) after approximately three minutes, after which the rate of growth slowed down somewhat because significant portions of the network did not have enough bandwidth to allow it to operate unhindered. Most vulnerable machines were infected within 10 minutes of the worm's release. Although worms with this rapid propagation had been predicted on theoretical grounds, the spread of Sapphire provides the first real incident demonstrating the capabilities of a high-speed worm. Even though Sapphire did not contain a malicious payload, it caused considerable harm simply by overloading networks and taking database servers out of operation. Many individual sites lost connectivity as their access bandwidth was saturated by local copies of the worm and there were several reports of Internet backbone disruption (although most backbone providers appear to have remained stable throughout the epidemic). It is important to realize that if the worm had carried a malicious payload, had attacked a more widespread vulnerability, or had targeted a more popular service, the effects would likely have been far more severe.

Figure 6 shows the spread of the sapphire worm within 30 minutes

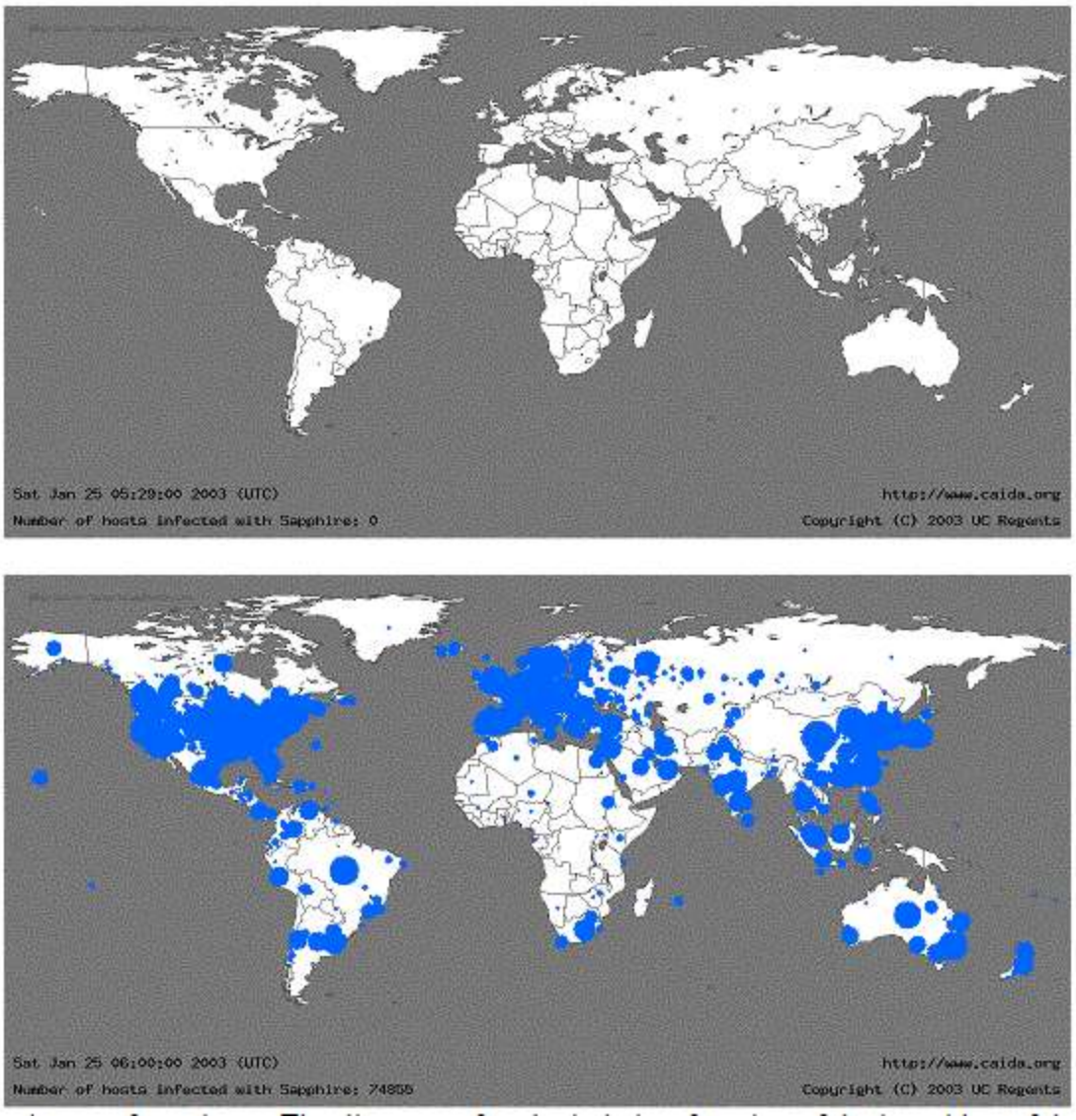


Figure 7 The geographic spread of Sapphire in the 30 minutes after release.

The diameter of each circle is a function of the logarithm of the number of infected machines, so large circles visually underrepresent the number of infected cases in order to minimize overlap with adjacent locations. For some machines, only the country of origin can be determined (rather than a specific city).

2.3.2 The Intersection of Big Data and Security Analytics

A research carried in 2013 by Enterprise Service Group (ESG) (The Enterprise Strategy Group, 2013) defines big data as a collection of data sets so large and complex that it becomes difficult

to process using on-hand database management tools or traditional data processing applications. The challenges include capture, processing, storage, search, sharing, analysis, and visualization.

The report shows the following as the characteristics of big data - the four Vs: volume, velocity, variety, and veracity.

Volume - When the term big data is used, data volume typically ranges multiple terabytes to petabytes. This certainly fits the enterprise security model as it is not uncommon for large organizations to collect tens of terabytes of security data on a monthly basis.

Velocity - This term is used with respect to real-time data analysis requirements. In cyber security, velocity can refer to the need for immediate anomaly, or incident detection. Real-time data analysis is critical here to minimize damages associated with a cyber security attack.

***Variety* -** Big data can be made up of multiple data types and feeds including structured and unstructured data. From a security perspective, data variety could include log files, network flows, IP packet capture, external threat/vulnerability intelligence, click streams, network/physical access, and social networking activity, etc. It is not unusual for enterprises to collect hundreds of different types of data feeds for security analysis.

Veracity - Big data must be trustworthy and accurate. From a security perspective, this means trusting the confidentiality, integrity, and availability of data sources like log files and external data feeds.

According to the ESG Research, 44% of enterprises (i.e., organizations with more than 1,000 employees) say that security data collection and analysis would be considered big data within their organizations today, while another 44% believe that they will likely consider security data collection and analysis big data within the next 24 months.

2.3.3 Large Organizations Need to Improve Security Analytics

Organic changes like data growth and longer retention periods are certainly major catalysts of big data security analytics as per the ESG report, nearly half of enterprise organizations (47%) simply believe that a big data security project is a logical progression to the amount of data collected/analyzed today

2.3.4 Hadoop Distributed File System (HDFS)

In the book entitled Hadoop, the definitive guide by Tom White, (White, 2012) the following have been listed as the benefits of HDFS:

- Reliable since there is data replication to many nodes thus improves fault tolerance.
- Supports parallel processing hence increases the speeds of computation.
- Relatively cheap to implement since it runs on commodity computes.
- Has been adopted by big IT companies such as Yahoo!, Facebook and Amazons and works well hence well tested.
- Provides very high throughput hence data access and is majorly designed for large data sets.
- Relaxes some POSIX requirements to enable streaming access to file system data.
- HDFS is part of the Apache Hadoop Core project hence have an open source license.

Apache Software Foundation (Apache Software Foundation, 2007) shows the following to be some of the desirable goals for the HDFS.

2.3.4.1. Hardware Failure

Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional.

Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

2.3.4.2. Streaming Data Access

Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.

2.3.4.3. Large Data Sets

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

2.3.4.4. Simple Coherency Model

HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A Map/Reduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future.

2.3.4.5. Moving Computation is Cheaper than Moving Data

A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to move the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.

2.3.4.6. Portability across Heterogeneous Hardware and Software Platforms

HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications.

2.3.5 NameNode and DataNodes in HDFS

HDFS has master/slave architecture.

An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients.

In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write

requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

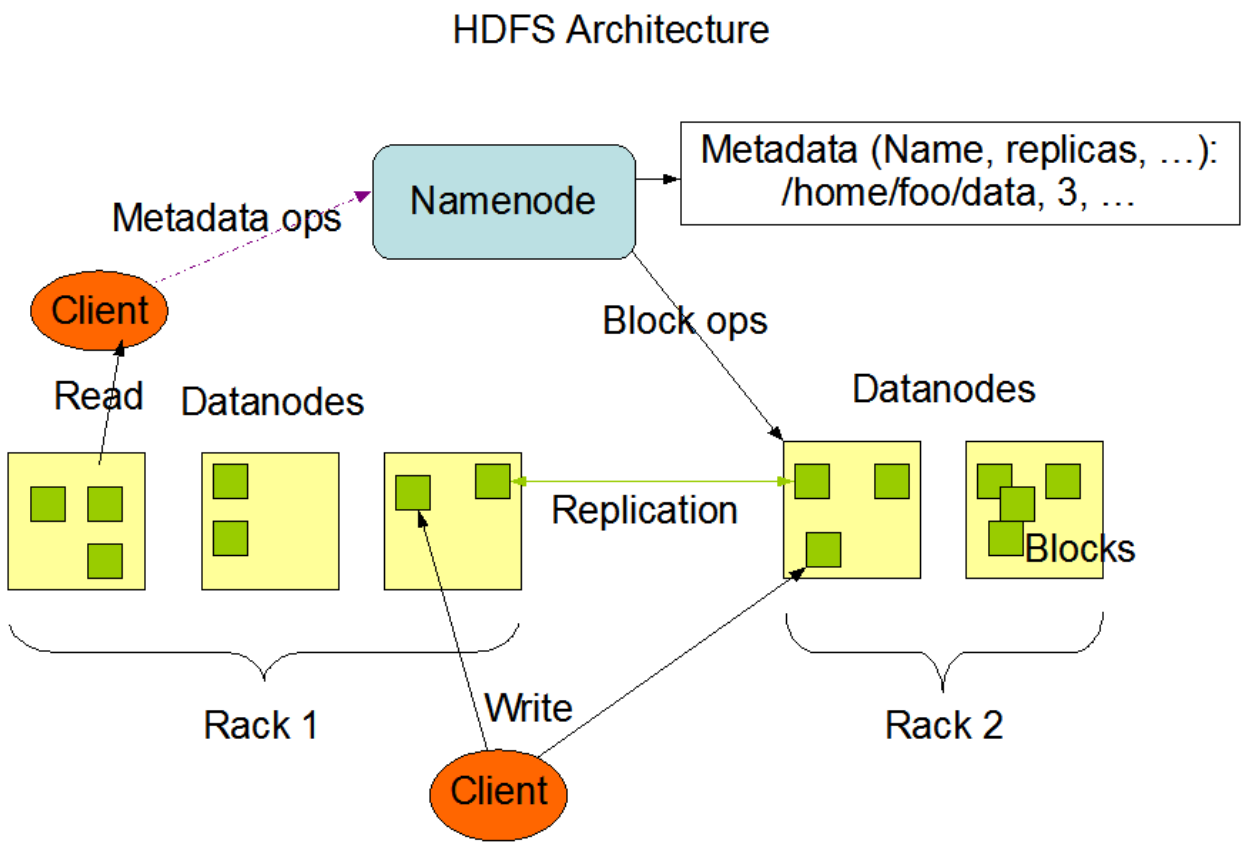


Figure 8 HDFS architecture

2.3.5 Big Data Technologies for Near-Real-Time Results by Intel

Intel Corporation (Intel Corporation, 2013) shows that balancing the compute, storage, and network resources in an Apache Hadoop cluster enabled the full benefit of the latest Intel processors, solid-state drives, 10 Gigabit Intel Ethernet Converged Network Adapters, and the Intel Distribution for Apache Hadoop software. Building a balanced infrastructure from these components enabled reducing the time required for Intel to complete a TeraSort benchmark test workload from approximately four hours to approximately seven minutes, a reduction of roughly 97 percent. Results such as this from big data technologies pave the way for low-cost, near-real-time data analytics that will help businesses respond almost instantly to changing market conditions and unlock more value from their assets.

2.3.6. Data Flow in HDFS

To get an idea of how data flows between the client interacting with HDFS, the namenode, and the datanodes, consider Figures 9 and 10.

(a) Reading a File in HDFS

Figure 3 shows the main sequence of events when reading a file. The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (step 1 in Figure 9). DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the cluster's network). If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block. The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O. The client then calls read() on the stream (step 3). DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream (step 4). When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order, with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls close() on the FSDataInputStream (step 6). During reading, if the DFSInputStream encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The DFSInputStream also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode before the DFSInput Stream attempts to read a replica of the block from another datanode. One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

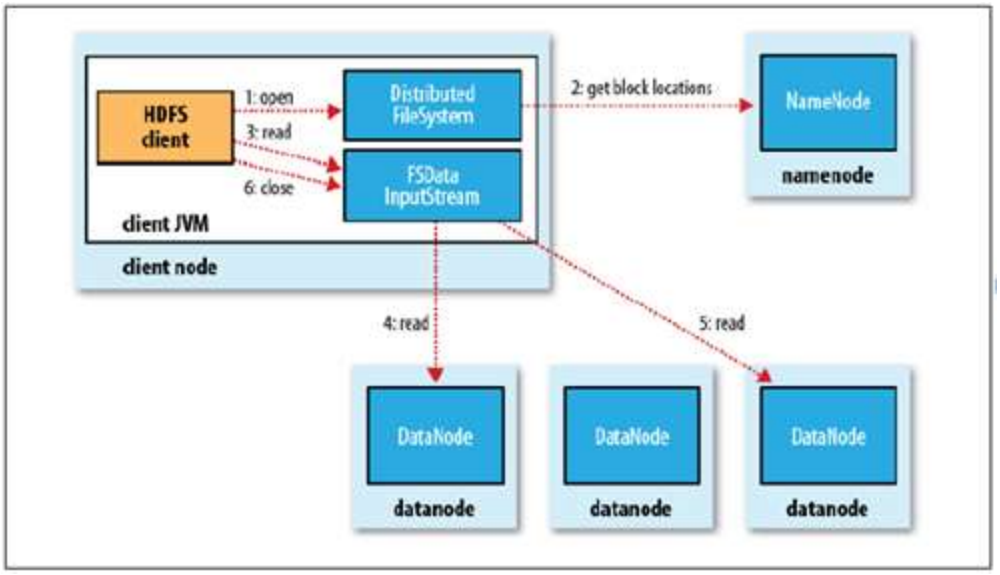


Figure 9: Client reading from HDFS

(b) Writting a File in HDFS

Next we'll look at how files are written to HDFS. Although quite detailed, it is instructive to understand the data flow because it clarifies HDFS's coherency model. We're going to consider the case of creating a new file, writing data to it, then closing the file. See Figure 4.

The client creates the file by calling create() on DistributedFileSystem (step 1 in Figure 4). DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException. The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode. As the client writes data (step 3), DFSOutputStream splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the Data Streamer, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ackqueue only when it has been acknowledged by all the datanodes in the pipeline (step 5). If a datanode fails

while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on. The failed datanode is removed from the pipeline, and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal. It is possible, but unlikely, that multiple datanodes fail while a block is being written. As long as dfs.replication.min replicas (which default to one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to three). When the client has finished writing data, it calls close() on the stream (step 6).

This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of (via Data Streamer asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

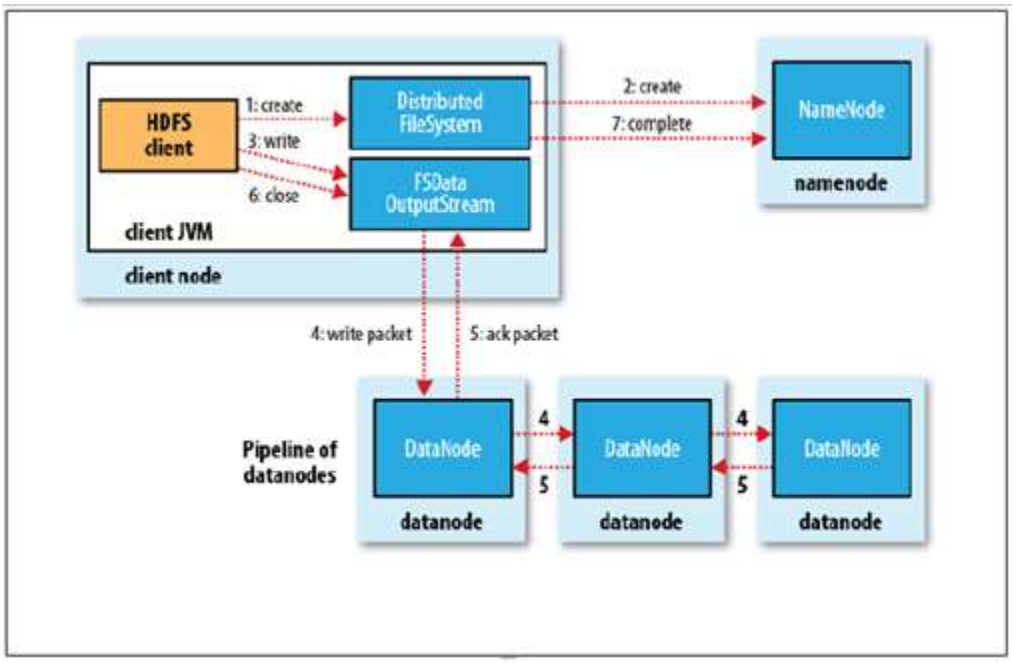


Figure 10: Client writing to HDFS

2.3.8. Technical overview of virtual machines

Virtual machines consolidated in one main server with architecture as shown below.

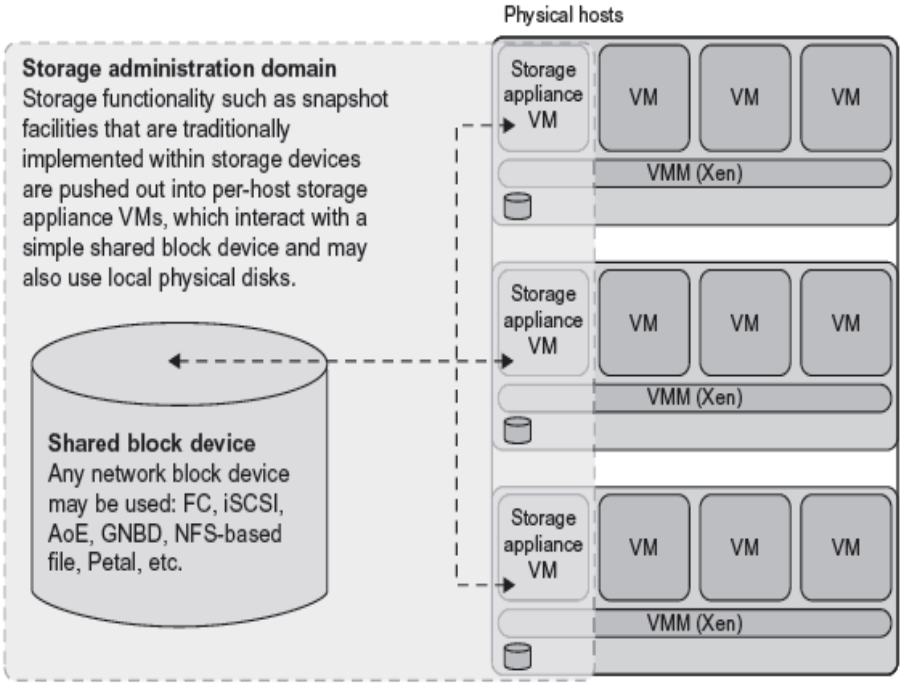


Figure 11 Server consolidation architecture

CHAPTER THREE: METHODOLOGY

3.0 Introduction

This section provides an introduction to the research methodology that was used in this research, its justification for use in the research, tools used and the model of the research.

3.1 Research design and its justification

This was an experimental study. An experimental study is a type of evaluation that seeks to determine whether a program or prototype had the intended causal effect on program participants.

In this research, we studied the impact of having more processing nodes on signature generation system in distributed systems.

3.1.1 Research design

This research was based on security analytics with the use of big data technologies using Hadoop Distributed File System and Map Reduce parallel computing programming model. It is meant to develop and evaluate a prototype for near real-time machine driven signature detection, generation and collection.

Two models of machine driven signature detection, generation and collection were created and compared against one another.

One model was based on a node computer (standalone model) while the other model was based on a network of multi-node of computers (the cluster model).

The standalone model exploited the techniques of sequential processing where the signature detection, generation and collection are performed in tandem, one stage completing before the other stage begins while the cluster model exploited the techniques of parallel processing, based on the Map Reduce distributed parallel programming model and improved reliability over Hadoop Distributed File System.

The specifications of the computers were held constant at 2GB RAM, 100GB HDD and 8 processing cores, across the study, also the amount of data being processed, also the amount of

data from which the signatures were to be generated while the number of processing nodes were varied from one test to the other.

The results were then compared with those of existing systems for signature generation in regards to time and accuracy.

3.1.2 Justification

The approach was chosen so that it would provide an opportunity to compare the differences in time taken to generate signatures over the same volume of data for the different programming models, so that the faster one would be adopted.

3.2 Tools for data collection and their justification.

3.2.1 Tools

The tools used for data collection were:

(i) Linux tcpdump package
(ii) Linux tshark package
(iii) PHP web crawler running on Linux OS command line interface
(iv) Wiereshark software
(v) PHP script for generating polymorphic versions of worm samples
(vi) Linux tmux package

3.3 Sources and methods for data collection and their justification.

3.3.1 Sources of data

Our primary source of data was network traffic.

We also generated some data using in-house programmed scripts.

3.3.2 Data collection methods and their justification

Our research required two types of data, namely the innocuous network traffic, which is the network traffic as observed during normal operation of a given network system and the malicious network traffic, which is the network traffic that is being suspected to be originating form an attacker.

Our principle was to get strings or substrings that occur so many times in the malicious flow, but not as much in the innocuous flow. Such strings are intuitively abnormal by virtue that they are not in the normal traffic and can be used to categorize or characterize the abnormal traffic. They are the strings we are looking for and can be used as security signatures by network equipment to guard against the aforementioned abnormal traffic.

Innocuous traffic was collected from HTTP traces of controlled web crawlers which we built and directed to known benign websites namely google.com, facebook.com, yahoo.com and php.net. This decision was so as to ensure that the data we were using as innocuous was actually innocuous since collecting web traffic from other sources such as actual human browsing would not guarantee that the data is free from any form of attack as we could not cannot control the sites that the actual users visit during their ordinary usage of the system.

Also this helped restrain the phenomenon called innocuous flow pool poisoning, a phenomenon whereby one deliberately adds noise to the normal traffic with intention to mislead the signature generator when that traffic is used in the signature generation process. In order to avert suspicious flow pool poisoning, a phenomenon where one adds meaningless traffic in the suspicious pool so as to mislead the generator into generating meaningless signatures, we ensure that some of the signatures must actually also occur in the innocuous traffic hence accept some degree of false positives which we strive to minimize. The collection was done behind a Linux tmux terminal for three days so as to avoid killing the processes when the ssh sessions were killed or terminated for any reason.

Malicious traffic was generated from known worm samples, where by worm requests as seen by network equipment were collected, and polymorphic versions of corresponding worms requests generated using PHP programming language, bearing in mind the invariant bytes of the polymorphic worms as well as the fillers. This decision was arrived at because there was no actual worm outbreak during the period of research and again it was desirable that the amount of worm samples in the particular dataset be known so that it could assist during analysis such as comparison of the accuracy of the prototype in regards to false positives and false negatives.

This would not be achieved with data captured from the wild since the actual composition would not have been known.

Two models of machine driven signature detection, generation and collection software were programmed using Java programming language with NetBeans 6.9.1 IDE.

One model was based on a single node computer while the other model was based on a network multi-node of computers.

The single node computer model exploited the techniques of sequential processing where the signature detection, generation and collection are performed in tandem, one stage completing before the other stage could begin while the multi-node model exploited the techniques of parallel processing, based on the Map Reduce distributed parallel programming model and improved reliability over Hadoop Distributed File System (HDFS).

Hadoop cluster was then set at the University of Nairobi School of Science laboratory where eight virtual machines of specifications 2GB RAM, 8 processing cores and 100 GB Hard Disk Drive were created on an Ubuntu server.

64-bit Centos 6.5 was then installed on all the virtual machines and the machines networked and configured to communicate with one another in a cluster environment and also access the Internet using public IPs provided by the University of Nairobi so as to enable installation of the cluster packages and also access the Internet for network traffic gathering.

The single node computer version of the developed software was deployed on one of the virtual machines while the distributed version of the software was deployed on the Hadoop Cluster for distributed processing.

The specifications of the computers were held constant across the study while the amount of data from which the signatures were generated was varied from one test to the other.

The collected network traffic was preprocessed using the aforementioned tools, to be particular, tshark and wireshark were used to extract only the payload sections of the network traffic from the collected data, in a new line delimited records so as to achieve structured network flows which were later fed into the signature generators and observation made.

Records were taken upon feeding varying amount of data and with different characteristics in regards to worm composition, considering the particular type of worm as well as the quantity of

samples of the given worm in the given dataset to both signature generators and observations with regards to time taken to generate the signatures, accuracy (false positives and false negatives) recorded. Also reliability was noted since obliterating the machine running the single node version meant the whole system went down while obliterating one machine in the cluster only reduced the performance of the overall system but at least it would generate some signatures.

The worms samples used were polymorphic version of Code Red II.

The following section shows the way data was collected and the table that was filled upon observations as different tests were being run.

A given volume of data was held constant and number of machines in the cluster was varied from one to eight, also the stand-alone machine version system was tested. The table below was filled as the tests were being carried out and data analyzed as shown in Chapter 6.

Size of the malicious dataset in GB _____Number of worm samples _____ COV_____

Size of normal dataset in GB _____ Number of Worm samples_____ FP _____

Number of Hosts	System Type	Time to accomplish processing	Average CPU Load	Average Memory Load	Signature generated
1	Stand-alone				
1	Cluster				
2	Cluster				
3	Cluster				
4	Cluster				
5	Cluster				
6	Cluster				

7	Cluster				
8	Cluster				

Table 3 Data results table

3.4 Data analysis methods and their justification

3.4.1 Data analysis methods

The aim of data analysis is to examine and organize data in a way that provides answers to research question and ensures that the research objectives are met. This process involved;

- Cleaning the data so as to correct errors and omissions.
- Sorting out data by arranging the data into different groups depending on characteristics of interest observed in the collected data
- Tabulation will involve summarizing raw data and displaying data in a format that further analysis can be undertaken.

Once necessary data had been collected a graph was drawn as shown in section 6.

Microsoft Excel program was used to perform the analysis.

3.4.2 Justification

These data analysis methods were chosen due to the simplicity of data that was collected and also the graphical outputs are very easy to interpret at a glance.

3.5 Limitations of methodology.

Poor implementation of an algorithm in terms of software engineering principles can undermine the performance of an algorithm and thus mislead results.

3.6 Assumptions made.

Hadoop Distributed File System is in itself secure and cannot be compromised during security monitoring.

CHAPTER FOUR: ANALYSIS, DESIGN AND IMPEMENTATION

4.0 Introduction

The high level architecture of our prototype is very similar to Lisabeth's, from which it is derived. In this section, we highlight the key features of the prototype

4.1 Global Overview of the prototype

Figure 4.1 shows the architecture of our prototype. We first need to sniff network traffic, reassemble flows of network packets and classify flows in terms of different protocols (TCP/UDP) and port numbers. For each (port-protocol) pair, we filter out known worm samples and then, using the worm flow classifier, we separate flows into suspicious (M) and innocuous (N) pools.

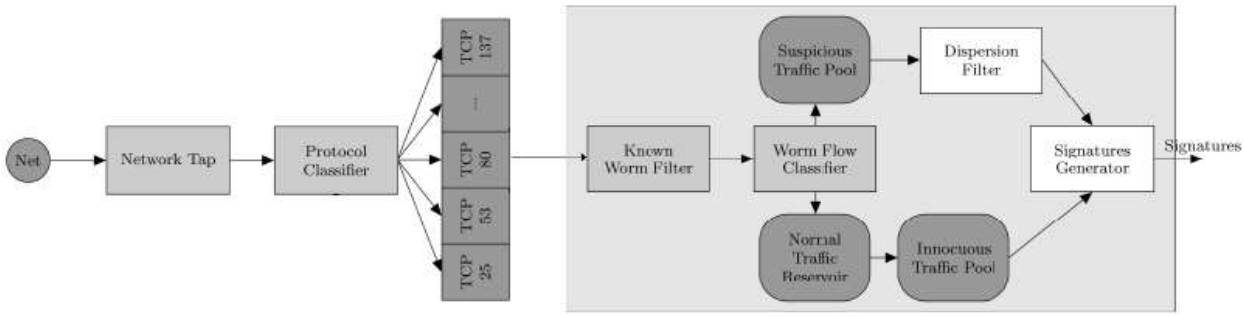


Figure 12 High level architecture of the prototype

The next step concerns the selection of suspicious and innocuous flows to send to the signatures generator and from which signatures will be created. To avoid poisoning attacks from a single attacker, i.e. with network packets coming from a single network address or at most from a limited network address set, we used a dispersion filter. The goal of this filter is to perform a dispersion analysis on suspicious flows in order to send to the generator a well dispersed set of flows. By using this technique only fewer flows for each source address are sent to the signature generator. Thus, to be successful, worm instances that want to perform suspicious pool poisoning attacks against our system must synchronize themselves with respect to the features to use in flows. This, as a consequence, makes the attack harder to carry out.

On the innocuous flows pool, instead, the idea is to use a good selection policy to decide which flows should be employed in the signature generation. Even if, as we will see in chapter six, our system is less sensible to innocuous pool poisoning attacks than Hamsa, we suggest the use of a dispersion-based flow selection policy also on the innocuous pool.

The selected suspicious and innocuous flows are given as input to the signatures generator which generates signatures as described in the following section.

4.2 Signature Generator

Unlike Hamsa, the only assumption our prototype is based on is that a true worm flow must contain all true invariants I_i of the invariant set I and that, in order to have a rapid spread, the worm sends worm samples across the network.

As it is possible to see in figure 4.2, the first operation performed on the suspicious flows pool is token extraction.

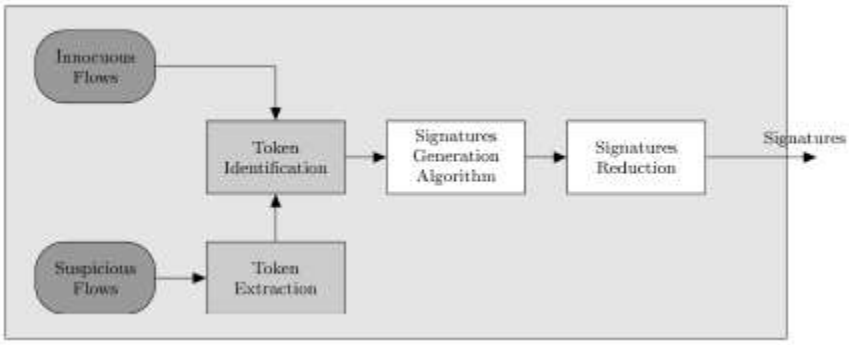


Figure 13 The signature generator module.

In this phase we find all sequences of at least ⌊ bytes that occur in more than λ fraction of the suspicious pool. The constraint on sequences length is required to ignore too small tokens, while λ is used to take into account only those that occur in several flows.

To speed up the algorithm's execution time, all the extracted tokens are then identified in each innocuous flow, and all flows, innocuous and suspicious, are converted in sequences of tokens discarding all bytes sequences not included in the extracted tokens set. Flows so obtained are then sent to the signature generation algorithm that, as we will see in the next, creates the required signatures.

The last phase performs signatures reduction on returned signatures to remove all tokens that, always appearing as subtokens of others ones, are not required to enhance signatures specificity.

4.3 Signature Generation Algorithm

Given suspicious (M) and innocuous (N) flows pools, the aim of the signature generation algorithm is to find a set S of signatures s_i such that $FP_{si} \leq FPmax$ and $COV_{si} \geq COV_{min}$. To this end Hamsa adopts a purely greedy approach. However, building the most specific signature including all possible invariants giving a good coverage of M without having any knowledge of the nature, real or fake, of the invariants or using a greedy algorithm with a restricted view of global situation without having a global overview of all possible signatures may weaken the strength of the detection, as we discuss in Chapter 5. To avoid generation of redundant signatures, we enforce an additional constraint in our algorithm:

$$s_i \in \mathcal{S} \Leftrightarrow \nexists s_j \in \mathcal{S} \mid \mathcal{M}_{s_i} \subseteq \mathcal{M}_{s_j}$$

The idea is to create all the signatures that match a considerable fraction of the suspicious pool, while avoiding the addition of new tokens to a partial signature when it has already an acceptable false positive rate (i.e. a rate less than FP_{max}).

In this way, we can have more signatures per worm but we have surely at least one specific enough signature containing only a subset of I. The value used for FPmax may be smaller than the value used in Hamsa for the shortest signatures, i.e., composed by only one token, and so the maximum false positives rate, accepted for a single signature, is lower.

The generation of all the possible subsets of the extracted tokens and subsequent check of the given constraints would require a non-negligible computational effort, so another aim of the algorithm is to avoid, whenever possible, to generate redundant or useless long signatures.

As we can easily think, for any given token there may be more occurrences of it in a single flow. To avoid worrying about this problem, we assign to each pair (token-number of occurrence) a unique identifier. In this way we consider these occurrences as different tokens. It is important to note that the same occurrence of the same token in different flows will have the same identifier.

All these matching information between identifiers and related (token;-number of occurrence) pairs are stored in an appropriate data structure, called TM, for subsequent use.

Once this identifier is assigned, for each of the aforementioned pair, we build a list of all suspicious flows in which it occurs, create a partial signature with that pair and related flows list and insert this new partial signature into the partial signatures set PS. To avoid generation of redundant signatures caused by the high number of tokens that appear always as subtokens of other ones, we remove these subtokens from PS and take them into account at the end of the signature generation algorithm. We say that t1 is a subtoken of t_2 if t1 $\neq t_2$, t_1 is substring of t_2 and the occurrence number considered for t_1 is the same of that considered for t_2.

Then, if a token t_1 occurs as a subtoken of t_2 we delete partial signature containing t_1 and add t_1 in the subtokens list of t_2 stored in subtokens data structure ST.

Once the partial signature set is build, our algorithm proceeds iteratively. First it evaluates the false positives rate of all the available partial signatures. Those with a value low enough are inserted into S and deleted from PS. The remaining signatures are then merged into each other and if each new partial signature has not a good coverage of M, it is discarded along with those of the prior iteration. Coverage evaluation is simple and fast: the number of covered flows is given by the intersection between flows lists of the two partial signatures merged together. The merging of two partial signatures is performed only if the new one has just one more token than the two from which it is obtained.

Finally, iterations are stopped when the partial signature set is empty. Before returning it, to each generated signature are added the ignored subtokens of each included supertoken.

Algorithm in appendix II describes in detail the generation algorithm developed to address the above requirements. The algorithm relies on the following methods' definitions as implemented Java programming language.

getTokenList() Returns the list of extracted tokens and, for each of them, the multi-set of flows in which it appears.

maxOcc(t) Returns the maximum number of occurrences of the token t in a single flow, considering all the suspicious flows in which the extracted token occurs.

genNewId() Generates a new unambiguous identifier.

checkIfSubT(t, i) Checks if the occurrence i of token t occurs as a subtoken of some supertoken.

FindSuperToken(t, i) Finds supertoken identifier for an occurrence i of the token t.

getFlowList(t, i) Returns the list of flows in which the occurrence i of token t occurs.

calcCov(e, P), calcFP(e, P) Return, respectively, the coverage and false positives of an element e on pool P.

merge(e, f) Returns a new element that contains the union of the tokens e and f and the intersection of their covered flows.

tokenNum() Returns the number of elements included into the token set on which is called.

newSign(e, TM, TS) Generates a signature containing tokens of e, all their subtokens, suggested by TS, and substitutes identifier of each token with the associated string, following TM hints.

4.5 Distribution for distributed/parallel processing

The algorithm in appendix II is meant to run on a standalone. To achieve distribution, it had to be broken into two parts, the map stage and the reduce stage of the Hadoop MapReduce Process. This was achieved by distributing the methods defined in section 4.4 in appropriate stages such that the following arrangement was achieved.

4.5.1 Map Stage Methods

The following methods as defined in section 4.4 were implemented in the map stage of the MapReduce process:

(i) getTokenList()
(ii) calcCov(e, P),
(iii) calcFP(e, P)
(iv) getFlowList(t, i)

4.5.2 Reduce Stage Methods

The following methods as defined in section 4.4 were implemented in the reduce section of the MapReduce process:

(i) maxOcc(t)
(ii) genNewId()
(iii) checkIfSubT(t, i)
(iv) FindSuperToken(t, i)
(v) merge(e, f)
(vi) tokenNum()
(vii) newSign(e, TM, TS)

4.6 Technical overview of the systems

Both systems are based on Lisabeth Signature generation models.

Figures 9 and 10 show the level 0 dataflow diagrams of the two models. The overview of the core differences between the two models can be seen. One is based on parallel computing while the other one is based on sequential computing.

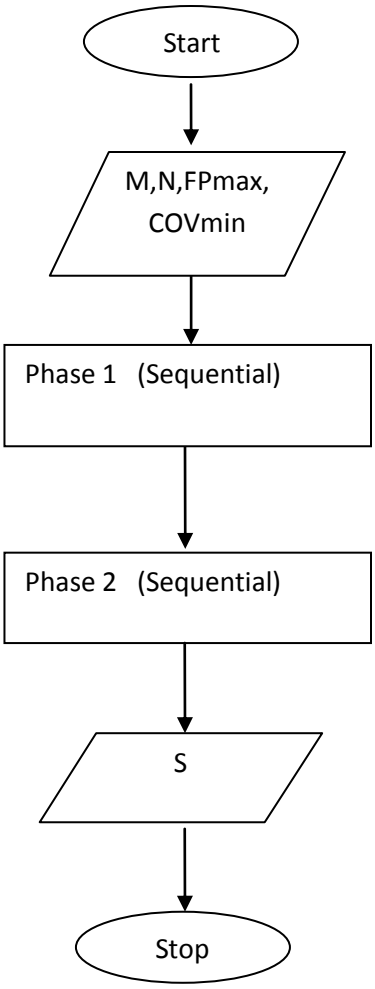


Figure 15 Single node model level 0 data flow diagram

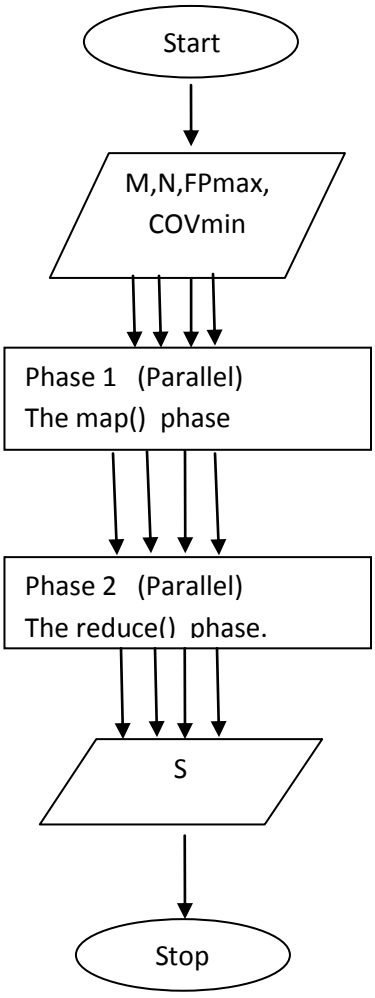


Figure 14 Multi-node model level 0 data flow diagram

4.7 System Functional Specification

This section describes the functional requirements of the signature detection, generation and collection system.

4.7.1 Functions Performed

The system takes in a pool of malicious network flow and outputs network security signatures out of it.

Three functions are performed.

(i) Detecting and extracting suspected security tokens a given malicious network flow pool

(ii) Generation or construction of security signatures, if any, from the extracted tokens

(iii) Outputting the security tokens for collection.

4.7.2 User Interface

Since the system runs as a daemon in the core network equipment, there is no need for graphical user interface. The system was being started from the command line and all the required arguments passed at the point of starting.

The signature output was collected from the predetermined output file.

4.7.3 User Inputs

The following inputs were provided by the users.

(i) The FP_{max} – this is the maximum acceptable false positive rate, given as a percentage value for example 0.1% or 0.001.

(ii) The COV_{min} – this is the minimum acceptable coverage that a given sequence of characters must have for it to be considered a signature token so it is taken for analysis. For example, a byte sequence may need to have to exist in 70% of the flows (defined as coverage of 70% coverage) for it to be extracted as a token.

It is the total number of flows that a given sequence of bytes exists, considering the number of occurrences per flow such that the same byte sequence occurring once in one flow and twice in the next flow are considered to be of different coverage.

(iii) The malicious flows pool.

This is a file containing network flows that have been classified as malicious, using various techniques such as deep packet inspection mechanisms or stateful packet inspection mechanisms.

The flows should be availed in the form of lines of records, each line in the file representing a network flow.

This input is constructed by absolute file path, for example:

C:\ signatures \MaliciousFlows.txt

(iv) The normal flow pool.

To prevent poisoning attack, where an attacker provides purely abnormal network flows to the system hence misleading signature generation process since it will base on statistics of useless flows, it is incumbent that we provide some normal pool which we use to base any incoming flows and discarding purely abnormal flows before reaching the signature generator. This is normal pool presented in form of a normal flow pool file and is given as an absolute file path, for example C:\signatures\NormalFlows.txt

4.7.4 System Data Base/File Structure

The system uses flat text files as its primary input data sources.

Here, we have a text file and every record in the file is represented by a line of text in the file, consequently, the records are delimited by the new line character.

The output is written directly to text files for collection and deployment.

4.7.5 System Performance Requirements

4.7.5.1 Efficiency (speed, size, peripheral device usage)

The distributed version runs on a cluster of computers to try and achieve improved processing speeds while the single version one, deemed slightly slower runs just on a single machine.

The system also fetches data from text files instead of relational databases so as to reduce the processing overheads that come with the relational database management systems.

Once data have been ingested into the system for processing, the system avoids writing back the data into secondary memory such as back to intermediary text files during processing, instead

holds all the data in Random Access Memory (RAM) during the whole processing period. This is expensive in terms of memory utilization but with the shared memory that comes with the cluster of machines, this is easily achievable and reduces the processing latency due seek time as this is ordinarily higher when data is stored in secondary memories.

CHAPTER 5: DATA ANALYSIS AND DISCUSSION

5.1 Data

Volume of data was held constant and number of machines in the cluster was varied from one to eight, also the stand-alone machine version system was tested. Data regarding processing times, memory utilizations, CPU loads as well as generated signatures were recorded as shown in table 4 below.

Size of the malicious dataset M in MB: 1000 Size of normal dataset N in MB: 100 Total M records: 50000 Number of worm samples in M: 30000 COV: 0.6 Total N records: 50000 Number of Worm samples in N: 500 FP: 0.01

Number of Nodes	System Type	Time to accomplish processing	Average CPU Load (%)	Average Memory Usage (%)	Signature generated
1	Stand-alone	∞	100	100	-
1	Cluster	∞	100	100	-
2	Cluster	33	78	100	{‘.ida’:1,’%u780:1,’HTTP/1.0\r\n’:1,”GET/’:1,’%u:2}
3	Cluster	20	69	96	{‘.ida’:1,’%u780:1,’HTTP/1.0\r\n’:1,”GET/’:1,’%u:2}
4	Cluster	15	62	82	{‘.ida’:1,’%u780:1,’HTTP/1.0\r\n’:1,”GET/’:1,’%u:2}
5	Cluster	11	54	70	{‘.ida’:1,’%u780:1,’HTTP/1.0\r\n’:1,”GET/’:1,’%u:2}
6	Cluster	9	47	61	{‘.ida’:1,’%u780:1,’HTTP/1.0\r\n’:1,”GET/’:1,’%u:2}
7	Cluster	7	39	53	{‘.ida’:1,’%u780:1,’HTTP/1.0\r\n’:1,”GET/’:1,’%u:2}
8	Cluster	6	30	38	{‘.ida’:1,’%u780:1,’HTTP/1.0\r\n’:1,”GET/’:1,’%u:2}

Table 4 Results of system evaluation.

5.2 Near real time speeds achievement.

Real time is defined as the exact time at which a particular activity takes place. In our context, we consider real time as the time when there is a worm outbreak and the worms are still spreading and attacking computers over the internet and hence signatures are desired in order to control it at that particular moment and reduce the associated costs that the spread comes with such as interruption of businesses.

The graph below shows our processing time vs. number of nodes

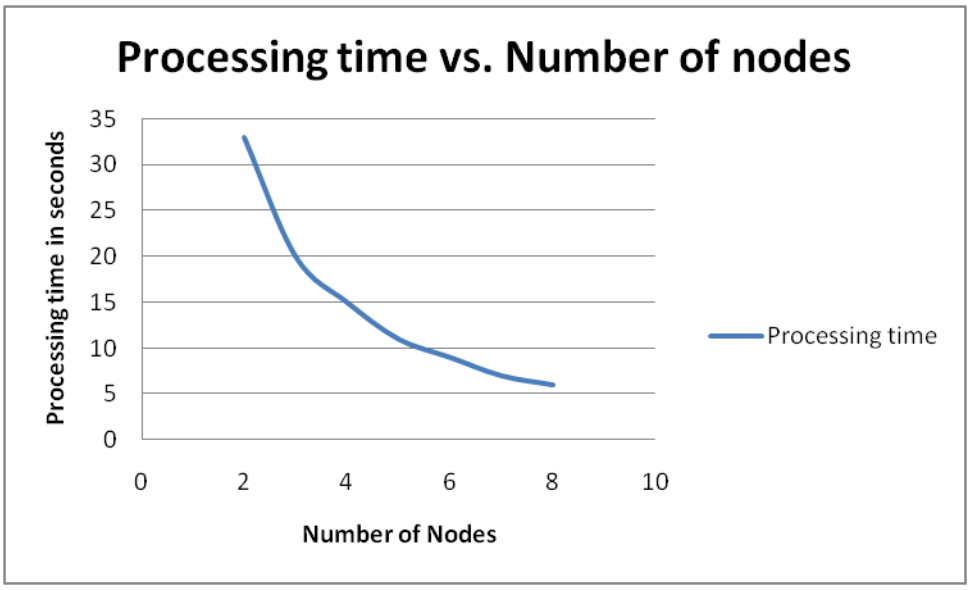


Figure 16 Processing time vs. number of nodes.

As can be seen, the amount of time required for processing a constant volume of data is inversely proportional to the number of nodes in the cluster.

In their research, (Moore et al., 2003) show that detection and containment must be initiated within minutes or seconds to prevent wide-spread infection in a 24 hour period.

According to Center for Applied Internet Data Analysis, (Center for Applied Internet Data Analysis (CAIDA), 2003) , the Sapphire worm was able to spread so quickly doubling the number of affected machines in every 8 seconds. Considering this as the maximum reaction time required to contain the worm, hence our allowable near real time, for the whole process of signature detection, generation and collection, we realize that this can be achieved by having a minimum of five nodes in the cluster, for our node specification of 2GB Memory, 8 processing

cores and 100GB Hard Disk Drive, below which it will be impossible to achieve near real time control of the worm.

Therefore the objective of achieving real-time speeds was attained in our research

5.3 Resource Utilization

5.3.1 CPU load

During the tests, the CPU load was also observed and data shown in table 4 recorded. For the cluster, we collected the average CPU load over the processing time for all the nodes and then performed an arithmetic mean, while for the standalone version of the system; the average CPU load for the VM running the single version was recorded. The graph was drawn as below.

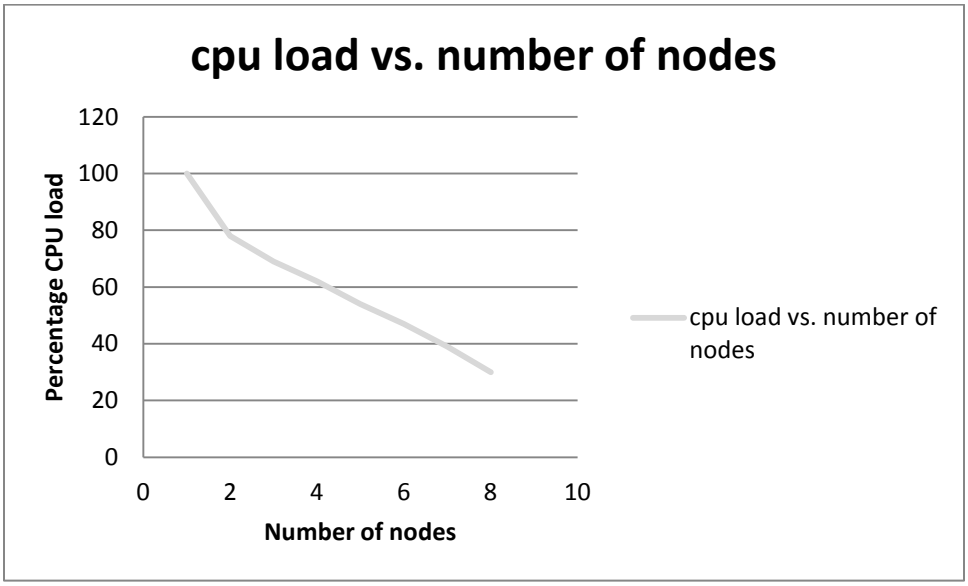


Figure 17 Cpu load vs. number of nodes

As can be observed, the CPU load is inversely proportional to the number of processing loads and as a result, it would be encouraging to add more processing nodes into the cluster in order to ease resource capping.

5.3.2 Memory Usage

Memory utilization of both the cluster and the standalone version of the system were also observed during the tests as recorded in table 4.

For the cluster, we recorded the average memory utilization of each node during the processing time then did an arithmetic mean of the results while for the standalone system, the average memory utilization over the processing time of the only node running the version was recorded.

The graph in figure 15 below was then plotted.

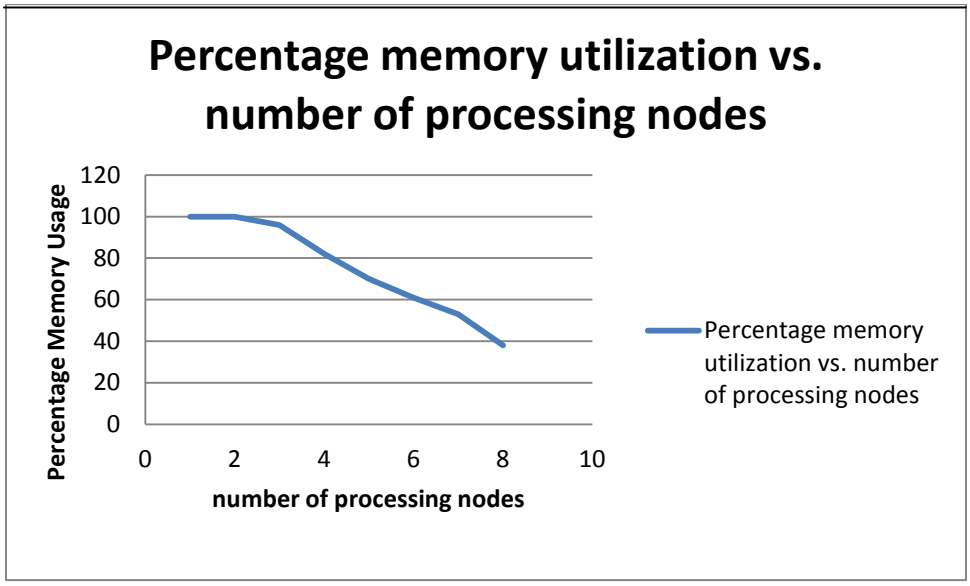


Figure 18 Memory usage vs. number of nodes

As can be seen from the graph, memory load is inversely proportional to the number of processing nodes hence to ease memory capping of the processing, one needs to add processing nodes

5.4 Accuracy

Accuracy is the degree with which a signature generation system generates correct signatures when there is an attack and generates no signatures when there is no attack. It is defined with false positives ratios and false negatives ratios as discussed below.

5.4.1 False Positives

False positive is the occurrence where by a signature generator generates a signature when there is no attack, i.e. it reports that there is signature (positive) when in actual sense there is no signature hence a false report.

To tame suspicious pool poisoning attacks, where an attacker sends so much useless traffic to mislead signature generation, it is always desirable that the traffic being sent for signature generation also exists in the normal network flow, meaning that the tokens extracted for

signature generation should also be seen in some percentage of the normal flow which ordinarily amounts to some deliberate degree of false positives which are inevitable.

False positives are calculated by a ratio of the worm flows in the normal flows to the total flows.

We can therefore afford to increase the size of the normal flow in the cluster environments as long as we increase the number of processing nodes and still afford generation of signatures within desired real time frame as desired.

Holding the number of worm flows in the normal flow in the normal traffic constant and increasing the number of the normal traffic will lead to lowered false positives hence improved accuracy and our system will still generate the signatures.

5.4.2 False Negatives

False negative is the occurrence whereby a signature generator fails to generate signatures when actually it is supposed to generate some signature i.e. it reports that there is no (negative) signature when actually there is a signature to be generated hence a false report.

As can be seen from our data, the standalone version of our system was overwhelmed with the processing of data, also the one node cluster and the whole process broke down, actually after investigation on the IDE why the process was stopping, we realized that the system was reporting an out of memory error as shown in appendix II after some time and then aborted the process before generating the signature. This means that if an attack was going on, whichever that would have been depending on the output would have thought that there is no attack and this would have been a false negative.

This was not the case with the distributed version especially with more than one node in cluster hence we can say that the increase in the number of nodes reduced the chances of false negatives due to the fact that nodes pooled together resources thus increasing memory as well as the processing power in the entire system in the cluster setup.

The reduction of false negatives correspondingly implies improved accuracy.

5.5 System Robustness

In computer science, robustness is the ability of a computer system to cope with errors during execution. In our case, we realize that in the cluster setup, if one node fails, the remaining nodes continue to work even though at reduced overall power. This means that signature generation process can continue to take place as opposed to a standalone system whereby when the only node running the entire system gets any mishap, the whole system stops the operation hence delicate.

5.6 Related Works

Even if early automated signatures generation systems (Kreibich & Crowcroft, 2003), (Kim & Karp, 2004) and (Singh et al., 2003) use different techniques to build worm signatures, all of them assume the presence of a single, specific enough, long invariant substring. Recently, there has been active research on polymorphic worm signatures generation, and new approaches have been proposed. New content-based systems like Polygraph and Hamsa have been deployed. As shown in this research, our system is very similar to these systems, but it is a significant improvement over Polygraph and Hamsa in terms of speed and attack resilience.

Behavior-based systems which use protocol and binary code information to characterize worm and subsequently build signatures, have been researched as well. Kruegel (Kruegel et al., 2005) propose an approach based on structural similarity of Control Flow Graph (CFG) to generate signatures for detecting different polymorphic worms. This approach, however, is computationally expensive and cannot detect worms with very small CFG or that apply special obfuscation techniques such as insertion of never exercised conditional branches. Of course, due to a more polymorphism resilience, it is also possible that this system detects worm that our approach misses. TaintCheck (Newsome & Song, 2005), working at host level, dynamically traces and correlates the network input to control flow changes to find the malicious input and derive worm properties. TaintCheck can understand worms and exploited vulnerability and it is able to automatically generate signatures.

CHAPTER 6: CONCLUSSION AND RECCOMENDATIONS

6.1 Conclusion

In this research, we have presented a near real time machine driven signature detection, generation and collection system for zero-day polymorphic worms. According to our experiments the prototype achieves significant improvements with respect to speeds over Lisabeth (Cavallaro et al., 2008), from which it is derived.

As can be seen from the analysis in chapter five, we were able to process more data at the same amount of time as compared to Lisabeth System by deploying the system in a cluster. Lisabeth System boasts of processing 100MB of data in six seconds while we were able to process a total of 1.1 GB in six seconds with a total of 8 nodes in the cluster.

Also as the trend of the graph shows, adding more nodes will lead to lower processing time and eases resources hence we have the allowance to continue reducing the speeds by adding more nodes.

This means that with our setup of specifications of 2GB RAM, 8 processing cores and 100GB Hard Disk Drive for every VM, we start realizing near realizing speeds after reaching seven nodes in the cluster, considering the 8.5 seconds propagation rate of the Sapphire worm, (Center for Applied Internet Data Analysis (CAIDA), 2003), below which we do not achieve the near real time speeds.

The ability to process more data gives room for distributed deployment of the system such as to have more points of inspections but still process the data in real time thereby denying the attacker an opportunity to evade a single point of inspection.

Regarding accuracy, as observed in chapter six, we can achieve lower false positives as we increase the number of nodes in the cluster since this enables us to process more innocuous flow, which when holding the number of worm samples in the normal flow constant as we increase the normal flow then FP correspondingly lowers since FP = (Worm Flows)/(Total normal flows) .

Still on accuracy, the chances of false negatives are reduced as can be seen that easing resources as seen in chapter six can lead to lower chances of system breaking down for instance like due to memory being overused as shown in appendix III when the systems break due to memory

shortage. Ordinarily this would cause a false negative since the system will break down before generating a signature and any other measures that depends on it for notification assume there was no signature to generate while actually the signature was there but the system went down.

Such scenarios are reduced by adding more nodes thus increasing the amount of resources in the cluster.

About robustness, we observe that the cluster can continue operating even on obliteration of one or more nodes even though performance will be reduced but at least we will achieve the mission of signature generation and depending on the initial size of the cluster, still we can achieve near real-time speeds, this cannot be the case with the single node version since any mishap to that node takes down the whole system.

Evaluating the performance of our system in terms of processing time, we can see that our system is able to produce the desired signatures in a desirable frame of time which based on the application it is intended for, is within real-time or near real time speeds.

6.2 Limitations

This section discus the limitations of the project

(i) Cluster instability.
One of the challenges with the project was the unstable nature of the set up cluster as shown in appendix V, whereby the cluster could run very well immediately after installation and later develop issues after some random duration.
This meant that we had to take so much time troubleshooting and eventually do a fresh install and run our tests at that point before leaving it again.

(ii) The tapping of network traffic for analysis consumes bandwidth hence adds a load to the monitored network.

(iii) Need to set up the Hadoop cluster may require additional operation cost from the implementing institution.

7.3 Recommendations for Future Works

However, we also have used basic string manipulation algorithms which exercises exhaustive search or full scan for token extraction which is seems slower and therefore recommends that future works adopt more efficient algorithms like enhanced suffix arrays for token extraction

References

Apache Software Foundation, 2007. *The Hadoop Distributed File System: Architecture and Design.* [Online] Available at: hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf [Accessed 14 June 2014].

Axelson, S., 2000. *Intrusion Detection Systems: A Survey and Taxonomy.* Technical Report 99-15. Sweden: Goteborg Chalmers University of Technology Department of Computer Engineering.

Bujlow, T., Carela-Espanol, V. & Barlet-Ros, P., 2013. *Comparison of Deep Packet Inspection (DPI)*

Tools for Traffic Classification. PhD Thesis. Bercelona: Polytechnic University of Catalonia Polytechnic University of Catalonia.

Cavallaro, L., Lanzi, A., Mayer, L. & Monga, M., 2008. *LISABETH: Automated Content-Based Signature Generator for Zero-day Polymorphic Worms*. Milan: University of Milan University of Milan.

Center for Applied Internet Data Analysis (CAIDA), 2003. *The Spread of the Sapphire/Slammer Worm*. [Online] Available at: http://www.caida.org/publications/papers/2003/sapphire/ [Accessed 14 June 2014].

Intel Corporation, 2013. *Big Data Technologies for Near-Real-Time Results*. [Online] Available at: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/big-data-apache-hadoop-technologies-for-results-whitepaper.pdf [Accessed 14 June 2014].

Kim, H.-A. & Karp, B., 2004. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Conference*. California, 2004. USENIX.

Kreibich, C. & Crowcroft, J., 2003. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *Second Workshop on Hot Topics in Networks (Hotnets II)*. Boston , 2003. Boston Publishers.

Kruegel, C. et al., 2005. Polymorphic Worm Detection Using Structural Information of Executables. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*. Seatle, 2005. WA.

Li, Z. et al., 2006. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *IEEE Symposium on Security and Privacy*. California, 2006. Oakland.

Moore, D., Shannon, C., Voelker, G.M. & Savage, S., 2003. *Internet Quarantine: Requirements for Containing Self-Propagating Code*. [Online] Available at: http://cseweb.ucsd.edu/~savage/papers/Infocom03.pdf [Accessed 14 June 2014].

Nazario, J., 2004. *Defense and Detection Strategies against Internet Worms.* Artech House.

Newsome, J., Karp, B. & Song, D., 2005. Polygraph: Automatically generating signatures forpolymorphic worms. In *IEEE Symposium on Security and Privacy*. California, 2005. Oakland Press.

Newsome, J., Karp, B. & Song, D., 2006. Paragraph: Thwarting Signature Learning by Training Maliciously. In *Ninth International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*. Hamburg, 2006. Germany Pub.

Newsome, J. & Song, D., 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *12th Annual Network and Distributed System Security Symposium*. Milan, 2005. University of Milan.

Paxson, V., 1998. Bro: A System for Detecting Network Intruders in Real-Time. In *7th USENIX Security Symposium*. San Antonio, 1998. Texas Publishers.

Roesch, M., 1999. Snort - Lightweight Intrusion Detection for Networks. In *In LISA '99 : Proceedings of 13th USENIX conference on System administration*. Berkeley, 1999. USENIX Association.

Singh, e.a., 2003. *The EarlyBird System for Real-time Detection of Unknown Worms*. San Diego: University of California.

Singh, S., Estan, C. & Varghese, G., 2003. *The EarlyBird System for Real-time Detection of Unknown Worms*. San Diego: University of California University of California.

Singh, S., Estan, C., Varghese, G. & Savage, S., 2004. Automated Worm Fingerprinting. In *Operating Systems Design and Implementation (OSDI)*. California, 2004. OSDI Press.

Staniford, S., Paxson, V. & Weaver, N., 2002. How to Own the Internet in Your Spare Time. In *11th USENIX Security Symposium*. California, 2002. Oakland Publishers.

The Enterprise Strategy Group, 2013. *The Evolution of Big Data Security Analytics Technology*. [Online] The Enterprise Strategy Group Available at: http://www.esgnext.com/c/701C0000000eY2H/pdf/ESG_MLR_Big_Data_Security_Analytics_Mar_2013.pdf [Accessed 14 June 2014].

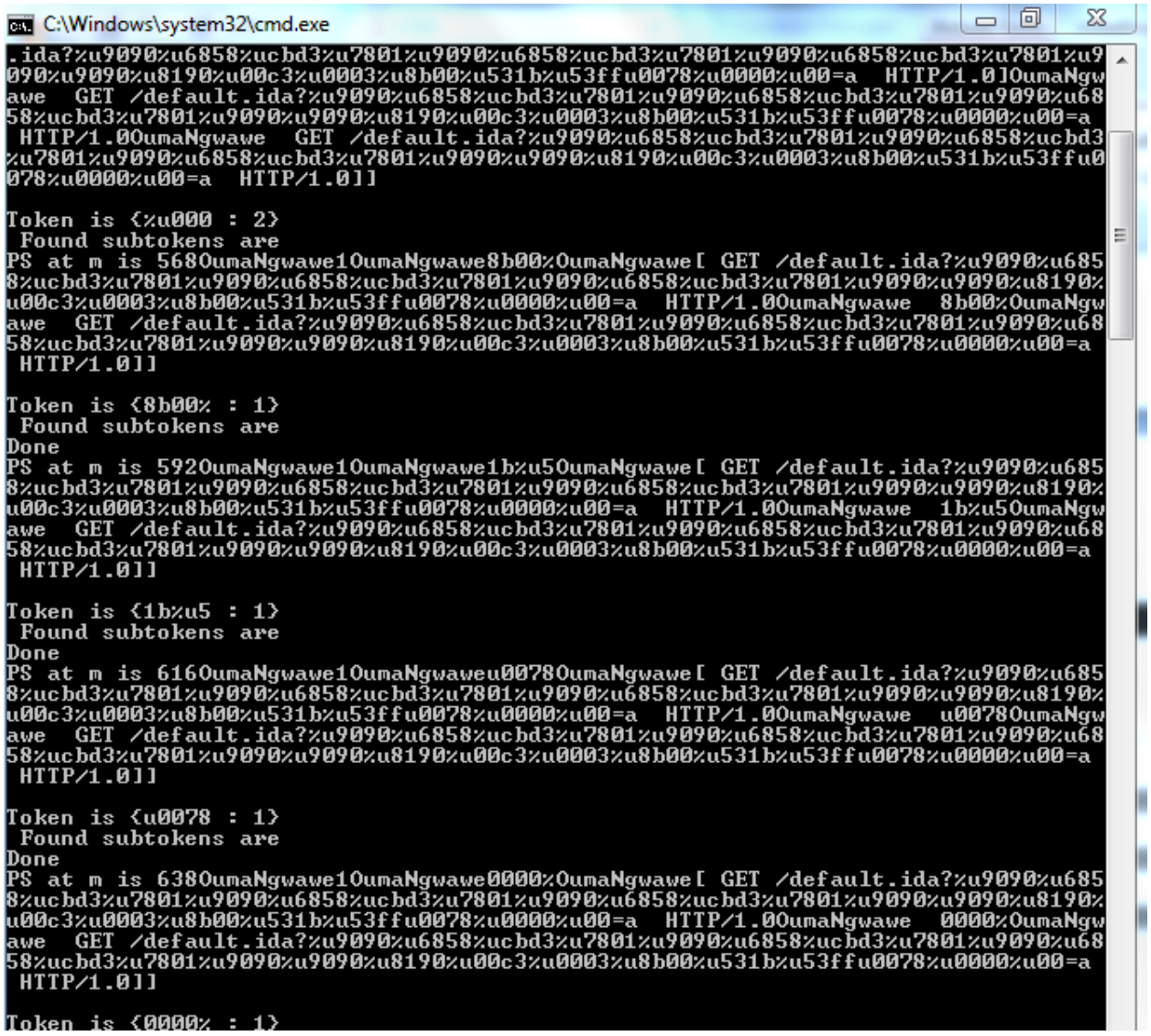
White, T., 2012. Hadoop: The Definitive Guide. In M. Loukides & M. Blanchette, eds. *Hadoop: The Definitive Guide*. 3rd ed. California: O'Reilly Media, Inc..

Wikipedia Inc, 2013. *Computer Worm*. [Online] Available at: http://en.wikipedia.org/wiki/Computer_worm [Accessed 20 June 2014].

Wikipedia Inc, 2013. *Real-time Computing*. [Online] Available at: http://en.wikipedia.org/wiki/Real-time_computing [Accessed 18 June 2014].

Zou, C.C., Gao, L., Gong, W. & Towsley, D., 2003. Monitoring and early warning for internet worms. In *10th ACM conference on Computer and communications security*. Washington D.C., 2003. ACM Press.

Appendix I Standard System Output on Command Line

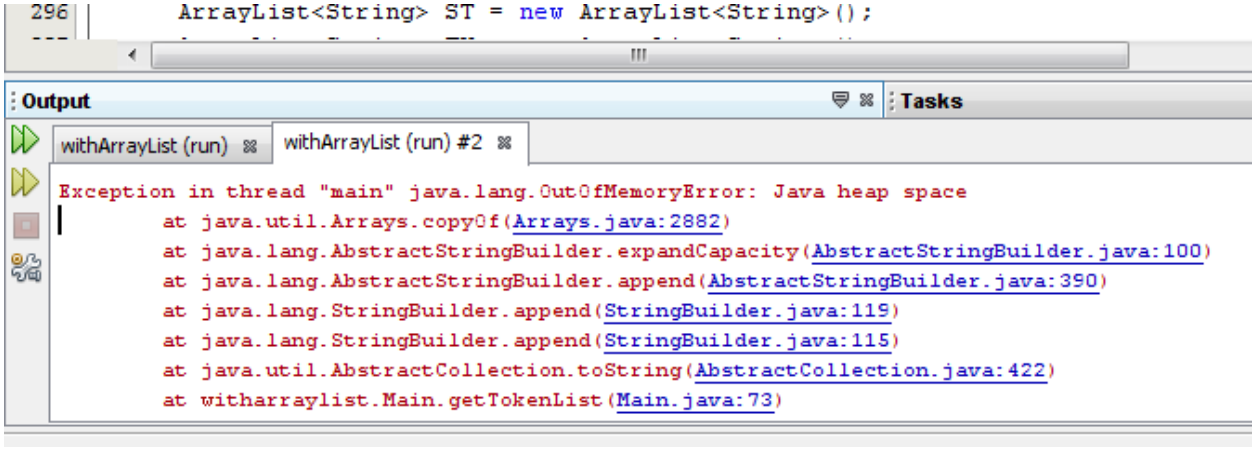


Appendix II: Lisabeth Algorithm

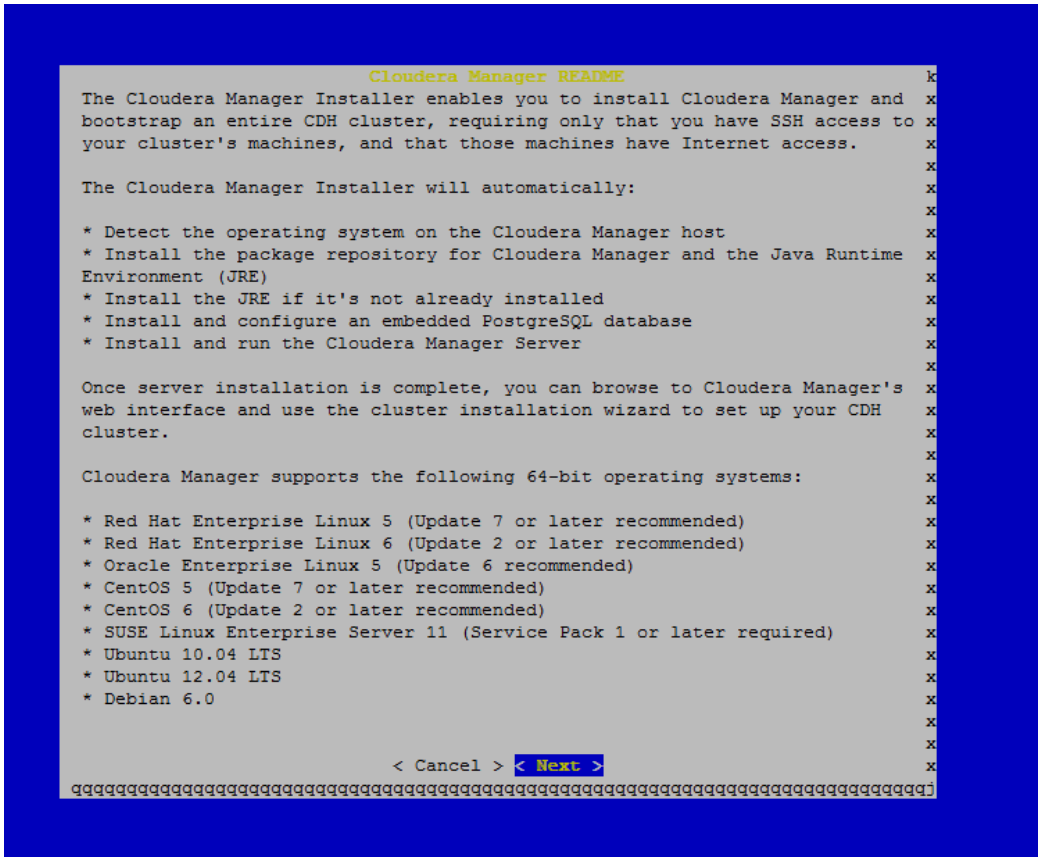
Input: $\mathcal{M}, \mathcal{N}, FP_{max}$ and COV_{min}
Output: Signatures set $\mathcal{S}$ for worms in $\mathcal{M}$

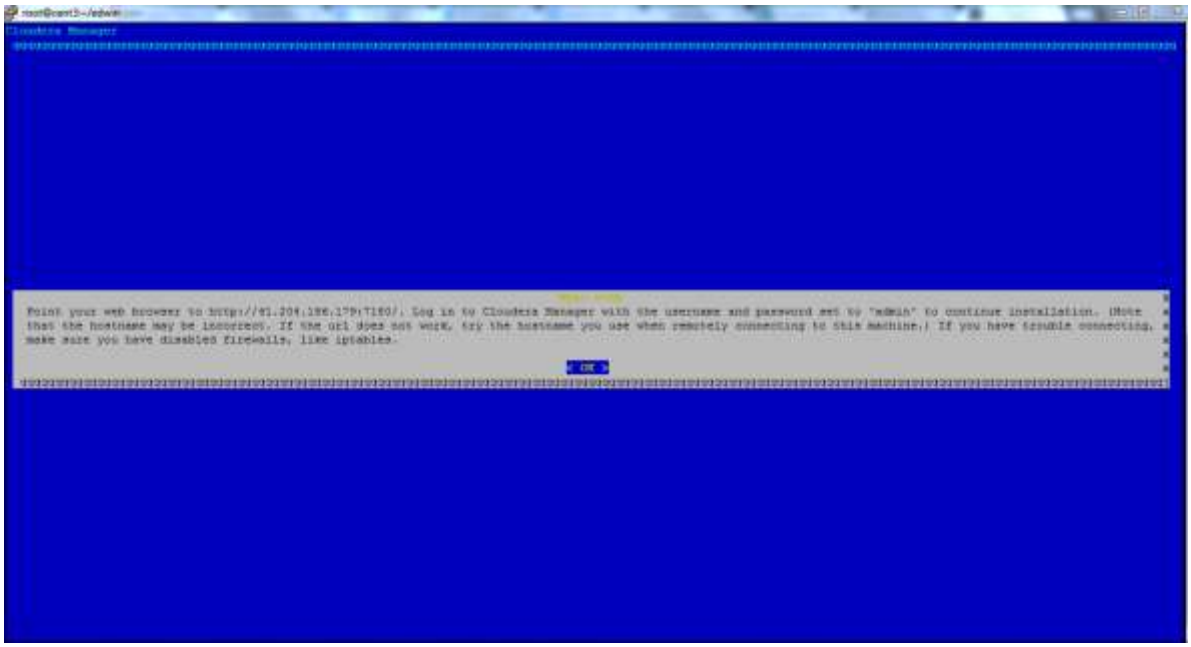
```
TM = PS = ST = S = ∅
tokenList = M.getTokenList()
tokenList.sort() {by descending length}
for all t ∈ tokenList do
  for i = 1 to tokenList.maxOcc(t) do
    id = genNewId()
    if PS.checkIfSubT(t, i) then
      ST.append(PS.findSuperT(t, i), id)
    else
      PS.append(genNewId(), id,-
      tokenList.getFlowList(t, i))
    end if
    TM.append(id, t, i)
  end for
end for
for all e ∈ PS do
  if PS.calcCov(e, M) < COV_min then
    PS.delete(e)
  end if
end for
signLen = 1
while PS.isNotEmpty() do
  signLen = signLen + 1
  for all e ∈ PS do
    if calcFP(e, N) < FP_max then
      S.append(newSign(e, TM, ST))
      PS.delete(e)
    end if
  end for
  for all e ∈ PS do
    for all f ∈ PS ∧ f.id > e.id do
      tmp = merge(e, f)
      if tmp.tokenNum() = signLen then
        if calcCov(tmp, M) ≥ COV_min then
          PS.append(genNewId(), tmp.id, tmp.flow)
        end if
      end if
    end for
    PS.delete(e)
  end for
end while
return S
```

Appendix III Memory Error Report

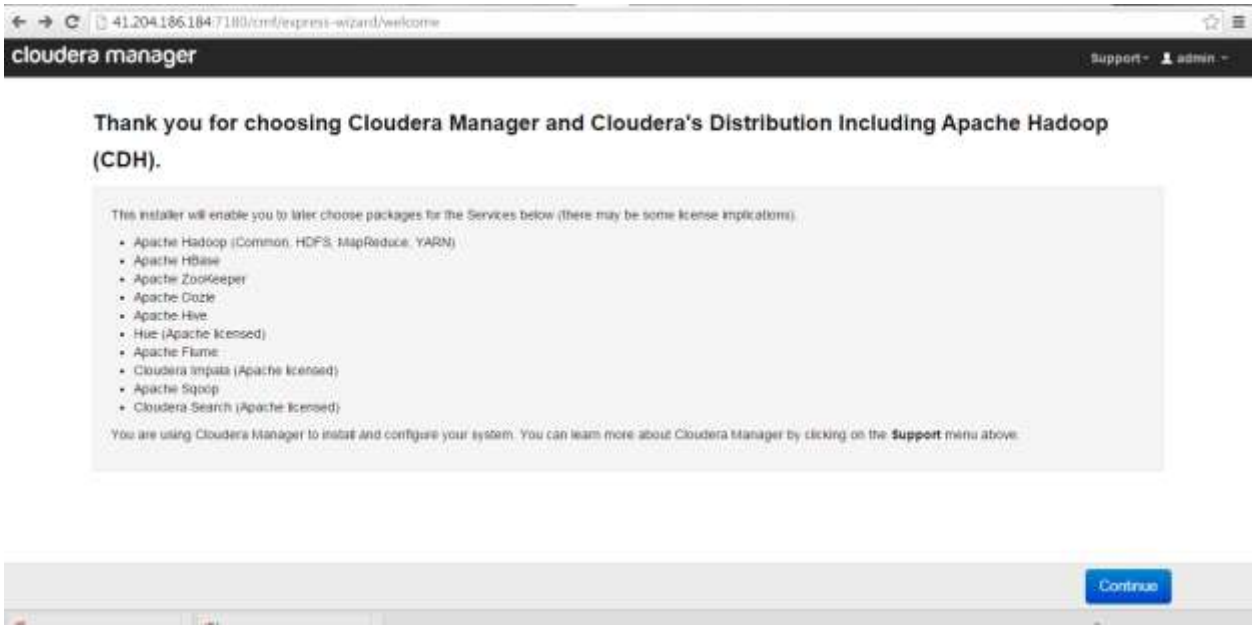


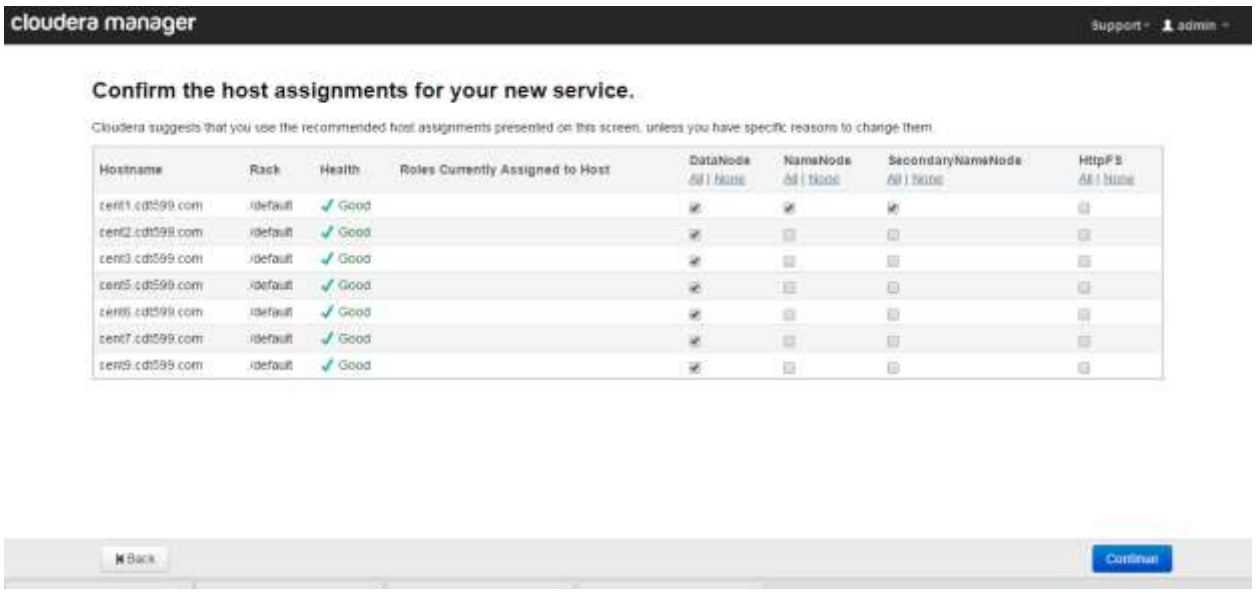
Appendix III – Installing Hadoop Cluster with Cloudera Package



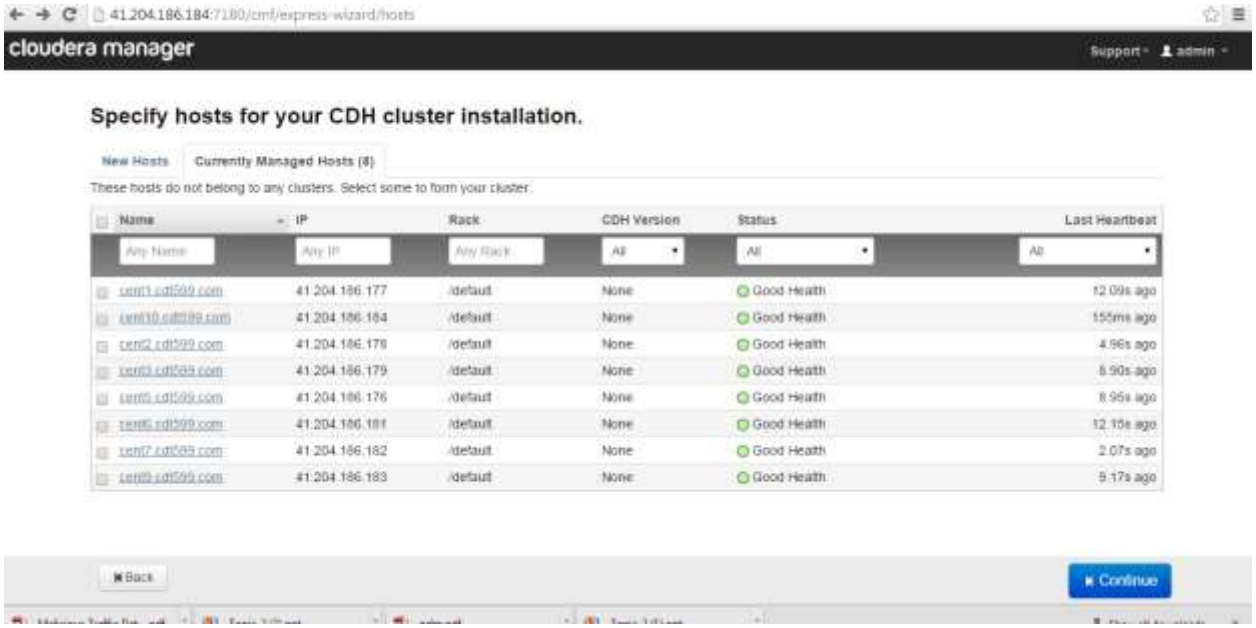


Appendix IV Installing Hadoop Services to Hadoop Cluster

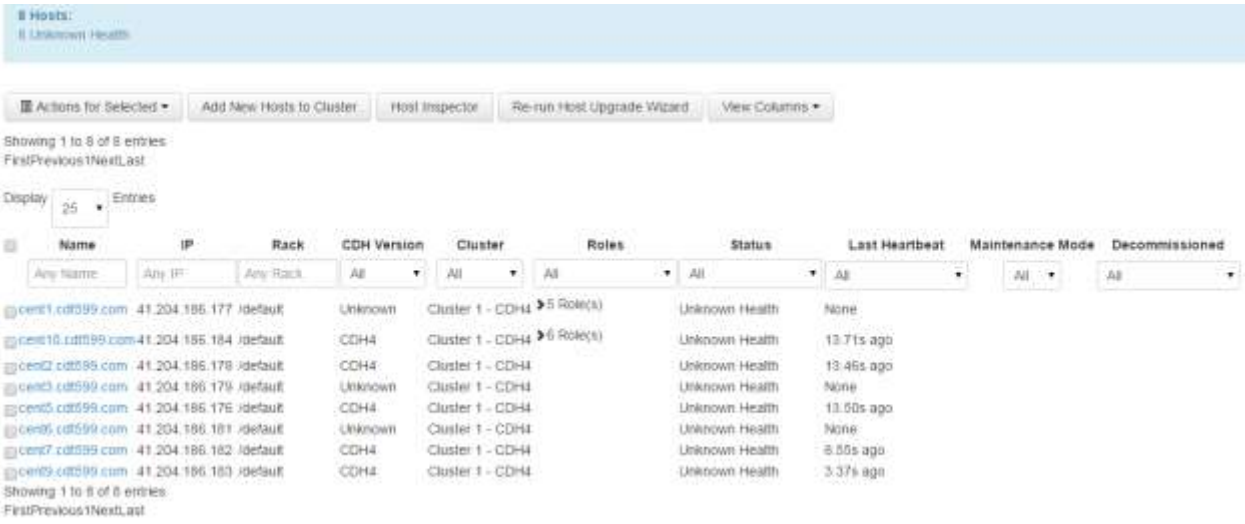




Appendix V Service Instability



After some time, without any human intervention, the health of the service becomes unknown.



Appendix VI Sample Code

```
package witharraylist;

import java.util.*;

import java.util.logging.Level;

import java.util.logging.Logger;

import java.io.*;

/**

 *

 * @author user

 */

public class Main {

   /**

    * @param args the command line arguments

    */

   public static ArrayList<String> getTokenList(String MaliciousFlowFile,int lmin, int lmax) throws FileNotFoundException{

      String tokenList = "C:\\edwin\\tokenList.txt";

      BufferedReader br = null;

      PrintWriter writer = null;

      String line = "";

      int l;

      int offset = 0;

      int rem;

      String NFile = "C:\\edwin\\testNormal.txt";

      String CT="";

      int Ms = 0;

      double COVmin = 0.1;

      double FPmax = 0.7;

      String flow ="";

      int cov =0;

      ArrayList<String> M = new ArrayList<String>();

      ArrayList<String> FlowList = new ArrayList<String>();

      ArrayList<String> PS1 = new ArrayList<String>();

      try {
```

```
   br = new BufferedReader(new FileReader(MaliciousFlowFile));
   try {
      while ((line = br.readLine()) != null) {
          M.add(line);
      }
   } catch (IOException ex) {
      Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
   }

} catch (FileNotFoundException e) {
                        e.printStackTrace();
         }
writer = new PrintWriter(tokenList);
writer.close();
for(l = lmax; l >= lmin; l-- ){ //1. iterate through all possible lenghts
     for(int c = 0; c<=M.size() - 1; c++){//2.iterate through all vaules of M (content)
     String content = M.get(c);
        rem = content.length()- (offset + l);
        for(offset = 0; rem >=l ; offset++){//3. iterate through all possible offsets that l can take in the given content
        CT = content.substring(offset,(offset + l));
        rem = content.length() - (offset + l);
        int occurence = Main.countSubstring(CT, content);
        FlowList.clear();
        Ms = 0;
        for(int m = 0;m <=M.size() - 1;m++){
        flow = M.get(m);
        cov = Main.countSubstring(CT, flow);
        if(cov == occurence)
          {
             CT = Main.escapeStorage(CT);
             FlowList.add(CT);
             flow = Main.escapeStorage(flow);
             FlowList.add(flow);
```

```
                Ms = Ms + 1;

              }

              }

                String token_flow = FlowList.toString()+"\n";

                if((Main.calcCov(CT, MaliciousFlowFile) >= COVmin) &&(Main.calcFP(CT, NFile) <= FPmax)){// check if of desired COV and FPMax
before accepting

                PS1.add(token_flow);

                }

            }

         }

        }

  return PS1;

  }
```