Mitigation of Insider Attacks for Data Security in Distributed Computing Environments

Santosh Aditham
University of South Florida, saditham@mail.usf.edu

Follow this and additional works at: http://scholarcommons.usf.edu/etd
Part of the Computer Sciences Commons

Scholar Commons Citation
Mitigation of Insider Attacks for Data Security in Distributed Computing Environments

by

Santosh Aditham

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Nagarajan Ranganathan, Ph.D.
Co-Major Professor: Srinivas Katkoori, Ph.D.
Adriana Iamnitchi, Ph.D.
Jay Ligatti, Ph.D.
Kandethody Ramachandran, Ph.D.
Babu Joseph, Ph.D.

Date of Approval:
March 7, 2017

Keywords: Big Data, Distributed Systems, Intrusion Detection, Control Flow, Statistical Analysis

Copyright © 2017, Santosh Aditham
DEDICATION

To my parents, sister and Dr. Ranga.
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Ranga for guiding and supporting me over the years. He is an example of brilliance as a researcher and mentor. But more importantly, he is the best example of persistence and tenacity.

I would like to thank my family for their help and support throughout this journey that wouldn’t be possible otherwise. My friends Sushma, Abhishek, Kalyan, Vishal, Raghav, Swetha, Sahitya, Ravi, Ravi Kiran, Pavitra, Shyam, Filipe, Ali Reza and Jamila have given their unconditional support as well. I would like to sincerely thank Dr. Katkoori for agreeing to be my co-major professor and for his unconditional support. Special thanks to Dr. Iamnitchi, Dr. Ligatti, Dr. Ramachandran and Dr. Babu Joseph for accepting to be on my committee and for their valuable suggestions.

I would also like to thank the Department of Computer Science and Engineering at University of South Florida and Amazon for providing me with the required financial support towards this research. Last but not the least, I would like to thank my teammates from Samsung, Juniper Networks and Apple for giving me the opportunity to intern with them and helping me in maintaining a balance between academia and industry. A special mention goes to Mrs. Radhika, whom I will always look up to.
# TABLE OF CONTENTS

LIST OF TABLES ........................................................................... iii

LIST OF FIGURES .......................................................................... iv

ABSTRACT ................................................................................. vi

CHAPTER 1 : INTRODUCTION ........................................................... 1
  1.1 Distributed Computing Environments for Big Data ................................ 1
  1.2 Privacy and Security of Data ..................................................... 3
  1.3 Motivation for this Research ..................................................... 4
  1.4 Contributions of Dissertation ..................................................... 12
  1.5 Significance of Contributions ..................................................... 12
  1.6 Outline of Dissertation ........................................................... 13

CHAPTER 2 : BACKGROUND AND RELATED WORK ................................... 15
  2.1 Frameworks for Big Data ........................................................ 15
  2.2 Target Attacks and Vulnerabilities ............................................... 16
    2.2.1 Insider Attacks......................................................... 16
    2.2.2 Programmer Errors ..................................................... 19
  2.3 Security in Distributed Systems .................................................. 20
    2.3.1 Intrusion Detection ..................................................... 20
    2.3.2 Tracing ...................................................................... 21
  2.4 Security in Big Data ............................................................. 23
    2.4.1 Attack 1: Data Loss .................................................... 25
    2.4.2 Attack 2: Data Degradation ............................................ 26
    2.4.3 Attack 3: Data Exposure ............................................... 28
    2.4.4 Attack 4: Infrastructure Degradation ................................... 28
  2.5 Special Hardware for Security .................................................... 30
  2.6 Control Flow Graphs ............................................................. 31
  2.7 Memory Access Patterns ......................................................... 35
    2.7.1 Principal Component Analysis .......................................... 36
    2.7.2 Long Short-term Memory ............................................... 37
  2.8 Context of this Research ......................................................... 38

CHAPTER 3 : PROPOSED FRAMEWORK ................................................ 40
  3.1 Local Analysis ................................................................... 40
  3.2 Dynamic Verification ............................................................. 41
  3.3 Secure Communication ........................................................... 41
  3.4 Model of the System Architecture .............................................. 44

CHAPTER 4 : COMPILATE-MI TECHNIQUES ............................................ 49
  4.1 Attack Model .................................................................... 49
4.2 Technique Based on Attack Probability Score ........................................ 50
  4.2.1 Proposed Methodology .................................................................. 50
  4.2.2 Experimental Results ................................................................. 59
  4.2.3 Analysis of Algorithm and Results ............................................. 62
4.3 Technique Based on Control Flow Graphs Analysis ........................... 65
  4.3.1 Proposed Methodology .................................................................. 65
  4.3.2 Experimental Results ................................................................. 69
  4.3.3 Analysis of Algorithm and Results ............................................. 73
4.4 Technique Based on Control Instruction Sequence Analysis .................. 73
  4.4.1 Proposed Methodology .................................................................. 73
  4.4.2 Experimental Results ................................................................. 79
  4.4.3 Analysis of Algorithm and Results ............................................. 84

CHAPTER 5 : RUN-TIME TECHNIQUES .................................................. 93
  5.1 Threat Model ........................................................................................ 93
  5.2 Technique Based on Memory Access Patterns and System Calls ........ 95
    5.2.1 Proposed Methodology ............................................................... 95
    5.2.2 Experimental Results ............................................................... 106
    5.2.3 Analysis of Algorithm and Results ........................................... 115
  5.3 Prediction of Attacks Based on Memory Access Patterns ................... 116
    5.3.1 Proposed Methodology ............................................................... 117
    5.3.2 Experimental Results ............................................................... 120
    5.3.3 Analysis of Algorithm and Results ........................................... 126

CHAPTER 6 : LIMITATIONS AND CONCLUSIONS .................................. 129
  6.1 Future Work ...................................................................................... 131

REFERENCES ........................................................................................ 133

APPENDIX A: COPYRIGHT CLEARANCE FORMS ............................... 144
LIST OF TABLES

Table 4.1  Basic Rule Engine ................................................................. 52
Table 4.2  Node Configuration ............................................................... 60
Table 4.3  Phoronix Test Suite Benchmark Results .............................. 63
Table 4.4  Program-Level Time Metrics of Hadoop MapReduce Examples (in Seconds) ...... 72
Table 4.5  Amazon EC2 Instance Types .................................................. 81
Table 4.6  List of Hadoop MapReduce Examples ................................... 82
Table 4.7  List of Spark-Perf MLlib Tests ............................................ 83
Table 4.8  Instruction-Level Properties of Hadoop MapReduce Examples .............. 85
Table 4.9  Instruction-Level Properties of Spark Performance Test ML Algorithms ........ 86
Table 4.10  Run Time Analysis to Analyze and Compare Hadoop MapReduce Examples ...... 88
Table 4.11  Run Time Analysis to Analyze and Compare Spark-perf MLlib Tests ............ 89
Table 5.1  F-Test Results on $t^2$ Statistic when Datanodes are Idle ............... 101
Table 5.2  Two Synthetic Intrusions for Testing the Proposed Solution ............... 108
Table 5.3  List of Hadoop MapReduce Examples ................................... 108
Table 5.4  Memory Properties of Hadoop MapReduce Examples ...................... 111
Table 5.5  Machine Configurations .......................................................... 121
Table 5.6  List of Hadoop MapReduce Examples ................................... 123
Table 5.7  Insider Attacks on a Datanode ............................................... 123
Table 5.8  RMSE Results from LSTM while Predicting Datanodes Memory Mappings .... 126
LIST OF FIGURES

Figure 1.1 Market for Big Data and Cloud Computing................................. 2
Figure 1.2 Relevant Attack and Vulnerability Statistics.............................. 6
Figure 2.1 Entities and Relationships in Insider Attacks............................ 18
Figure 2.2 A Scenario for Data Loss in a Hadoop Cluster.......................... 27
Figure 2.3 A Scenario for Data Degradation in a Hadoop Cluster.................. 29
Figure 2.4 Multiple MSAs for a Single CFG from Disassembled Object Code..... 33
Figure 2.5 Sample LSTM Cell Design [1]............................................. 38
Figure 3.1 System Architecture for Security in Big Data Systems.................. 42
Figure 3.2 Basic Hardware Model for the Proposed Security Framework.......... 46
Figure 4.1 Steps for Calculating Attack Probability Score.......................... 53
Figure 4.2 Communication Protocol in the Proposed Framework................... 57
Figure 4.3 Time Overhead for APS Technique....................................... 64
Figure 4.4 Proposed Algorithm for Intrusion Detection Using Process Signatures 66
Figure 4.5 Execution Time Analysis of CFG Technique............................. 71
Figure 4.6 Steps Involved in Attack Detection using CIS Technique............... 77
Figure 4.7 Consistent Distribution of CFI in Hadoop and Spark Tests............. 87
Figure 4.8 Run-time Analysis of Hadoop and Spark Programs....................... 90
Figure 4.9 Forecast Plots (Best-fit Regression) - Time to Detect an Attack...... 91
Figure 5.1 Memory Behavior of Datanodes in a Hadoop Cluster.................... 99
Figure 5.2 Runtime Intrusion Detection using Proposed Framework ..................... 105
Figure 5.3 Datanode Metrics Analysis for Teragen After Attack (source: Amazon EC2) .... 110
Figure 5.4 Datanode Metrics Analysis for Terasort After Attack (source: Amazon EC2) ..... 110
Figure 5.5 Analysis (ANOVA) of Cluster Behavior After a Replica was Compromised. .... 112
Figure 5.6 Analysis (Tukey) of Cluster Behavior After a Replica was Compromised........ 113
Figure 5.7 Cluster-level Call Frequencies After a Replica was Compromised by Attack 1.... 114
Figure 5.8 Cluster-level Call Frequencies After a Replica was Compromised by Attack 2.... 115
Figure 5.9 Steps for Attack Prediction using LSTM Technique........................... 118
Figure 5.10 Datanode Memory Behavior (Before and After Attack). ...................... 124
Figure 5.11 LSTM Analysis of Memory Behavior of a Datanode for MapReduce Examples ... 125
Figure 5.12 LSTM Root Mean Square Error Loss Convergence............................ 127
ABSTRACT

In big data systems, the infrastructure is such that large amounts of data are hosted away from the users. Information security is a major challenge in such systems. From the customer’s perspective, one of the big risks in adopting big data systems is in trusting the service provider who designs and owns the infrastructure, with data security and privacy. However, big data frameworks typically focus on performance and the opportunity for including enhanced security measures is limited. In this dissertation, the problem of mitigating insider attacks is extensively investigated and several static and dynamic run-time techniques are developed. The proposed techniques are targeted at big data systems but applicable to any data system in general.

First, a framework is developed to host the proposed security techniques and integrate with the underlying distributed computing environment. We endorse the idea of deploying this framework on special purpose hardware and a basic model of the software architecture for such security coprocessors is presented. Then, a set of compile-time and run-time techniques are proposed to protect user data from the perpetrators. These techniques target detection of insider attacks that exploit data and infrastructure. The compile-time intrusion detection techniques analyze the control flow by disassembling program binaries while the run-time techniques analyze the memory access patterns of processes running on the system.

The proposed techniques have been implemented as prototypes and extensively tested using big data applications. Experiments were conducted on big data frameworks such as Hadoop
and Spark using cloud based services. Experimental results indicate that the proposed techniques successfully detect insider attacks in the context of data loss, data degradation, data exposure and infrastructure degradation.
CHAPTER 1: INTRODUCTION

Cloud computing is defined as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [2]. It provides a swarm of services such as storage, software, platform and infrastructure. The huge adoption of cloud computing led to the advent of a new terminology called big data that is typically used to describe large volumes of structured and unstructured data. Businesses such as software, finance, telecommunications, retail, medicine and healthcare are all interested in uncovering the information hidden in big data. The euphoria around analyzing big data led to the start of a new interdisciplinary field called data science which is a mix of data analytics and machine learning. Data science tools implement and support algorithms that run on big data and they need to produce results almost instantaneously to handle the high demand and extreme competition. Inherently, an environment that can store, manage and process big data quickly and efficiently is one of the most sought-after resources today.

1.1 Distributed Computing Environments for Big Data

Distributed Computing Environment (DCE) is a software technology for setting up and managing computing and data exchange in a system of distributed computers [5]. Important components of DCE are: naming service, authentication service, timing service, communication protocol and file system. Though DCE has been around since the 90s, they became hugely popular in the
(a) Top Industries using Big Data in 2016 [3].

(b) Market Share of Public Cloud in 2016 [4].

Figure 1.1: Market for Big Data and Cloud Computing.
last decade after big data architectures started representing them. DCE is suitable for big data solutions because of their requirement to store petabytes, exabytes and larger amounts of data, and process terabytes of data. Hence, big data solutions are built as a software stack that runs on top of distributed computing environments. Today, big data solutions are used in various government and enterprise domains such as software, finance, telecommunications, retail, medicine and healthcare. According to a recent report by International Data Corporation [3], some of the frequent use-cases of big data are information retrieval from complex, unstructured data; and real time data analytics. Top computer industries such as Google, Amazon and Microsoft provide public and private clouds that serve as infrastructure for other companies to manage their data. It is projected that the worldwide revenues for big data and business analytics will grow from 130.1 billion dollars in 2016 to more than 203 billion dollars by 2020. Figure 1.1a shows the top industries that have already adopted big data solutions and Figure 1.1b shows the top public cloud providers that provide services to host big data solutions. These figures clearly suggest that the industry is moving towards cloud based big data solutions. Some of the popular big data frameworks are: Google file system along with MapReduce [6, 7], Hadoop [8], Spark [9].

1.2 Privacy and Security of Data

Along with its huge popularity and rapid market growth, the big data trend also has its share of challenges and risks. Initially, architects and developers of big data solutions focused only on performance. However, big data characteristics like velocity, volume, and variety magnify security and privacy issues. End users have to trust the providers of cloud and big data services that host their data. Such trust is built on an underlying assumption that the services will never be compromised. But unexpected issues such as insider attacks or exploitation of vulnerabilities due
to programmer errors can happen in any system anytime. In an era where extracting information from data is sanctioned to all, users are understandably more skeptical to let the service providers control their data that is stored away from them. This, along with the continuous increase in the number of cyber attacks, elevate the importance for security in big data. Figure 1.2a shows different kinds of vulnerabilities in big data related products from the Apache Foundation. Big data unicorns such as Hortonworks, Cloudera, IBM, SAS acknowledge the importance for security and privacy. Yet, the losses due to vulnerabilities in existing systems seem to overshadow the investments made towards increasing system security. This shows that big data security is still in its budding stage. For example, there is an immediate need to address architectural loopholes within the big data systems. Instead, extensive analysis of stored data continues to be the central point when talking about security and big data. In a privacy-first world, such models indirectly facilitate the abuse of user data in the hands of the service provider.

1.3 Motivation for this Research

There has been an unprecedented rise in malicious programs and attacks all over the world, as shown in Figure 1.2b. Institutions ranging from big companies such as LinkedIn, Yahoo, Target, Sony Pictures etc., to small businesses have all been targeted by attackers. Recently, two unauthorized backdoors were discovered in Juniper Networks firewalls that might have given attackers access to highly classified information. Some important facts about this particular hack are: (a) it comes at the cost of compromising national security (b) it shows that even a major network security company is vulnerable to attacks (c) it is believed that these backdoors were left undiscovered for almost 3 years knowing the high stakes and having vast resources; and (d) it was reported that the attackers could have deleted the security logs [10]. This is one of the many examples to show that
the common attack prevention techniques, such as identity management, access control lists and data encryption, are necessary but not sufficient to detect or prevent attacks. As per OpenSOC [11], in 60% of breaches data gets stolen within hours of the breach and 54% of breaches are not discovered for months. Another recent poll [12] revealed that 87% of companies reported being concerned or very concerned about data privacy in the cloud. Cyber attackers are able to successfully hack into the networks of small, medium and large organizations everyday irrespective of the numerous security systems in place. According to another recent survey [13], the top perceived security threats are 1) Unauthorized access (63%), 2) Hijacking of accounts (61%), and 3) Malicious insider attacks (43%). This indicates that software services need to have efficient attack detection techniques along with strong attack prevention techniques for robust security. This forms the main motivation for this dissertation research dealing with malicious insider attacks.

In this dissertation, the focus is on mitigating insider attacks that target exposure of data that is hosted on distributed computing environments. Insider attacks have become increasingly common and they are considered the toughest attacks to detect [16]. There does not exist much in the literature on solutions for insider attacks in general [17]. This has been discouraging customer companies from making use of cloud based solutions. Though privacy and security are touted to be important problems in the big data world, the solutions concentrate only on leveraging big data services for efficient security in other industries. To the best of our knowledge, there is no robust solution for detecting or preventing insider threats within big data infrastructures. For example, security mechanisms of popular big data systems such as Hadoop [8] and Spark [9] include third-party applications such as Kerberos [18], access control lists (ACL), log monitoring and data encryption (to some extent). But for an insider, especially a traitor, circumventing these mechanisms
Figure 1.2: Relevant Attack and Vulnerability Statistics.
is not difficult [19]. It is crucial to address the problem of insider attacks in big data systems for four main reasons: (a) traitor within the provider’s organization will be able to circumvent most of the security protocols in place (b) sensitivity of customer data stored in the system is increasing by day (c) a vulnerability or an attack will impact the confidentiality, integrity and availability guarantees given to the user; and (d) there is no consensus or widespread agreement on well-defined security standards in the big data community.

Intrusion attacks cover network intrusions, buffer overflows, protocol specific attacks such as man-in-the-middle attacks, trojans, insider attacks etc. Insider attacks are used to describe attacks that originate within an organization and are performed by employees of that organization who have legitimate access to the system, and knowledge about the development and working of the system. Typically deal with an employee stealing data using USB drives or by masquerading as another employee to gain access to unauthorized data [19]. Another form of insider attacks in big data is data degradation where an insider modifies user data. A side affect of insider attacks in big data is infrastructure degradation over time. Any such attacks are hard to detect and even harder to prevent. With the increase in popularity of concepts such as differential privacy, the biggest concern for these platforms is permanent data loss. Security compromised data often can be irreversible. Hence, big data solutions need to be able to identify an attack on the data as soon as it happens. In this regards, we believe that big data platforms might not need brand new security algorithms but they need existing security methods to be applied in different combinations and with new emphasis on securing user data and handling insider attacks. The main focus in this dissertation is to identify such methods, modify them according to the platform needs and test them thoroughly before proposing to use them.
The intent of security solutions is to detect malicious attacks before they compromise the security of the data. In this regards, solutions to insider attacks can be implemented at static compile-time and/or dynamically at runtime. Having compile-time security solutions will decrease the scope of possible insider attacks at runtime. This is very crucial for performance oriented systems such as big data applications because they cannot afford to cater too many resources at runtime. Compile-time analysis of programs is a static method to obtain information that can be used for several purposes such as enhancing parallelism and resource management. Some examples of such static analysis techniques in security domain are symbolic execution and control-flow integrity. However, compile-time analysis for improving security in big data has not been done before. Hence, in this dissertation multiple compile-time analysis techniques for attack detection are proposed. These techniques are tailored for verifying the control flow of a job scheduled to run on a big data system.

Typically in a big data cluster, when a user submits a request, the namenode (master) creates a job and schedules it for execution on the datanodes (slaves) that host the required data for the job. When the job is scheduled to execute at a datanode, static analysis techniques can be run on the associated compiled binary or bytecode to find vulnerabilities and bugs. In this regard, the proposed compile-time techniques address some of the data security concerns in big data platforms by performing analysis of compiled programs at assembly language level. These static analysis methods for intrusion detection can help mitigate the effects of vulnerabilities caused due to misplaced jumps, uninitialized arguments, null pointers, dangling pointers etc.

However, such static analysis techniques have their limitations. Vulnerabilities due to buffer-overflows, shared libraries and dynamic linking will continue to exist even after static analysis.
Memory corruption attacks can be detected and prevented at runtime with the help of address sanitizers [20]. But handling improper calls due to programmer errors is still a difficult problem to address. Insiders and masqueraders in distributed environments such as cloud and big data cannot be completely mitigated by using only static techniques [21]. Also, static analysis of programs will not work if a datanode configuration is changed by an insider or if a rogue datanode intentionally masquerades the information it shares. Due to the distributed nature of big data platforms and their requirement to provide data consistency (with the help of replication), it is possible to perform dynamic analysis of processes for attack detection at runtime and prevent adverse outcomes such as data loss. Hence, tighter intrusion detection and prevention techniques that analyze memory access patterns and other system properties of processes are also proposed in this dissertation.

Memory access pattern of a process correspond to the addresses referenced and the number of pages used. They typically remain the same for a given data used by the same application on hardware using the same instruction set and main memory. Hence, such patterns can be monitored and controlled by the system memory modules. For example, in many video and image applications, the data is stored in a specific sequence and distributed to memory modules in a fixed manner. A memory access pattern has multiple features to it such as shared pages, private pages, resident set etc. Each feature conveys some insight about the memory behavior of a process. But raw values of memory features can be diverse. Hence, Principal Component Analysis (PCA) is used in this work to find regression among memory features. Using PCA, raw memory access patterns can be represented as a T-squared distribution by calculating distance from the center of the translated space. Then, statistical analysis methods such as ANOVA and Tukey can be applied on such distributions for analysis and comparison of variance.
As big data technologies are becoming common, blind trust in the providers of big data platforms is essential if the end user wants to store their data in the cloud because such trust is built on an underlying assumption that the platforms and their security methods will never be compromised. As a result, the providers have de facto control over user privacy at all times. When unexpected issues such as insider attacks or vulnerability exploitation of programmer errors happen, the system typically relies on intrusion detection techniques every time. As such, techniques for intrusion detection is a research area that is constantly evolving [22, 23, 24]. However, in this dissertation we are also interested in developing a novel technique to predict the possibility of an attack. The motivation behind prediction of attacks is to limit the impact of an attack and to reduce the turnaround time of detecting attacks. Prediction of attacks involves analysis of behaviors and symptoms such as deliberate markers, meaningful errors, preparatory behaviors, correlated usage patterns, verbal behavior and personality traits [25]. From these sets of indicators, clues can be pieced together to predict and/or detect an attack. In preemptive protection of data, prediction techniques help in being prepared for an attack with predetermined security techniques.

Deep learning is a branch of machine learning based on algorithms that attempt to model high level abstractions in data. It can be interpreted as automated machine learning that trains on data to make predictions. Deep learning based neural networks prove to be efficient tools in predictive analytics because they can model data that contains non-linear characteristics and often in big data applications, data exhibits such non-linear properties. Hence, using deep neural networks to predict attacks in big data systems is a promising direction to go forward for security researchers. Long short-term memory (LSTM) is a popular recurrent neural network that has been proved to work very well for time series and other simple sequence predictions. This method is
widely used across domains such as image & video analysis [26], natural language processing [27], stock markets and anomaly detection [28]. Typically, LSTM networks are built to run on big data clusters but in this work LSTM networks are used to predict attacks within a big data cluster.

In the big data world, it is considered that moving computation to where the data resides is better than the traditional approach of moving data for computation. The main features of big data infrastructures are fast data processing, high scalability, high availability and fault-tolerance. Availability and fault-tolerance of big data systems rely on intelligent replication of data. This implies SPMD (single program, multiple data) style, parallel execution of the same program at multiple locations. When a program is scheduled for execution on the big data cluster, it runs as an individual process on every data node that hosts a copy of the program data. Data replication in big data is crucial to provide high availability and fast processing. This replication of data on various nodes in the big data system can also be utilized in providing security. Security for a computing system can be implemented at hardware and software level. Security provided at hardware level ensures isolation and tamper resistance of sensitive data such as keys used in cryptography. Given the advantage of isolation that can be achieved at hardware level security, we propose delegating security to special purpose hardware, such as Trusted Platform Module (TPM) [29] and Intel’s Trusted Execution chips (TXT) [30], that reside on the nodes of the big data cluster. Such an infrastructure will have the advantages of (a) performing security analysis remotely (b) reducing the overhead on main processor by delegating security, and (c) decreasing cost of data transfer while providing security.
1.4 Contributions of Dissertation

This dissertation explores some of the insider threats that exist in the big data field today such as data loss, data degradation, data privacy and proposes techniques to counter them. This is achieved by identifying specific areas in program behavior that pose threat to user data and understanding distributed properties of big data frameworks. The specific contributions are:

- An independent framework and system architecture for enhancing security in big data systems.

- A set of compile-time techniques for detection of insider attacks:
  - by estimating attack probability score of a program [19].
  - by checking for control-flow graph isomorphism [31].
  - by using control instruction sequences [32].

- A run-time technique for detection of insider threats using process information such as system calls and memory access patterns, and applying statistical comparison tests [33, 34].

- A run-time technique for predicting insider threats using memory access pattern of a process and applying deep learning [35].

1.5 Significance of Contributions

The first contribution of this work is an independent framework that can host multiple security techniques as needed by distributed computing environments. The software architecture of this framework is designed such that it does not hinder the overall performance of the system. This is possible because the proposed framework sits on top of the big data system. The proposed light-weight security framework can be deployed on special purpose hardware such as coprocessors.
that are dedicated for security. Because the proposed framework does not involve modification of underlying system, it can be adopted by any data system.

The effectiveness of a security framework depends on the security techniques it can accommodate. In order to protect user data from various insider attacks and threats, both compile-time and run-time security techniques are needed. For this purpose, a set of compile-time techniques are proposed in this dissertation to help with mitigating insider attacks that cause data loss and data corruption. This is achieved by analyzing the control-flow of the jobs that are scheduled to run on user data. These compile-time security techniques reduce the attack vector for local attacks that target data exploitation.

Once the stage of compile-time analysis is crossed, the processes running on the big data system are then analyzed by the proposed run-time security techniques. While one run-time technique proposed in this dissertation analyzes memory access patterns and system calls made by a job to detect attacks, the other technique works towards predicting the possible threats by analyzing the same memory access pattern of a process. A combination of all the compile-time and run-time techniques proposed in this dissertation will prevent insiders from successfully attacking the system.

1.6 Outline of Dissertation

The remainder of this dissertation is organized as follows: Chapter 2 describes the background and related work in the areas of security in general, security in distributed systems, security in big data, hardware based security, control flow graphs and memory access patterns. This chapter also gives a primer on Hadoop, PCA and LSTM. Chapter 3 presents software framework and the system architecture that are capable to independently host the proposed security techniques and
communication protocol. Chapter 4 presents a set of compile-time techniques proposed to detect attacks and Chapter 5 presents the two run-time techniques proposed to detect and predict attacks. The limitations of this work, conclusions and future work are presented in Chapter 6.
CHAPTER 2 : BACKGROUND AND RELATED WORK

Research topics closely related to this work are: (a) handling vulnerabilities caused by programmer errors, (b) protection from insider attacks, (c) intrusion detection techniques in distributed systems, (d) tracing application behavior for security in distributed systems, (e) security in big data platforms such as Hadoop, (f) understanding control flow graphs for process analysis, and (g) observing patterns in memory accesses for finding anomalies in process behavior. But first, a primer on big data frameworks is important to understand the necessity and impact of the aforementioned research areas.

2.1 Frameworks for Big Data

A plethora of big data frameworks are available today. Most of them are open-source under the Apache license, such as: Hadoop, Falcon, Atlas, Tez, Sqoop, Flume, Kafka, Pig, Hive, HBase, Accumulo, Storm, Solr, Spark, Ranger, Knox, Ambari, ZooKeeper, Oozie, Metron etc. These frameworks are usually hosted on distributed computing environments. Throughout this dissertation Hadoop [8] is used interchangeably for a big data framework running on a distributed compute environment. Thus, it is important to understand how Hadoop operates. The two major components of Hadoop are Hadoop distributed file system (HDFS) and MapReduce. HDFS serves as the distributed file system used to store massive data sets and MapReduce is used for data processing on HDFS. Hadoop follows a master-slave architecture to mitigate costs related to data processing, exception handling and attack recovery. Namenode acts as master of storage (HDFS)
and manages all metadata related to the file system. Datanodes are slaves of Namenode and are used to store the actual data, in blocks. JobTracker acts as a master of processing (MapReduce) and manages the jobs and their access permissions at cluster and user levels. TaskTrackers are slaves of JobTracker and are used to complete the map or reduce jobs allocated by their master. More recent versions of Hadoop have another major component called YARN for resource management. Namenode, datanode, jobtracker, tasktracker and all YARN components are all daemons that run on a cluster of machines. Together with all the services provided by the Hadoop ecosystem, this cluster of machines is called a Hadoop cluster. The basic workflow of Hadoop has 4 steps: HDFS write to load data into the cluster; MapReduce to analyze data in parallel; HDFS write to store results in cluster; and HDFS Read to read results from cluster.

2.2 Target Attacks and Vulnerabilities

This work considers only a subset of attacks and vulnerabilities that impact security of data in distributed computing environments that host big data. Insider attacks and other data attacks possible due to programmer errors and infrastructure loopholes make the attack vector of this dissertation.

2.2.1 Insider Attacks

Though security in general computing has been extensively studied and implemented over the years, computers are still vulnerable to attacks. Software based attacks that typically target a computer network or system, called cyberattacks, are growing in their frequency and impact. The plot for any type of software attack involves exploitation of a piece of code that runs on a computer. It is inherent to this perspective about a cyberattack that security can be provided
at two levels: (a) by the software that is used to compile and execute the program; and (b) by the hardware that runs the program. Providing security at software level gives more context and information about the target programs that are being protected. But this comes with the risk of the security software itself being compromised. On the other hand, having security at hardware level gives more isolation to the process of analyzing and securing programs though it becomes difficult to give detailed context about the programs and the infrastructures running them. In any case, the toughest software attacks to counter are the ones whose genesis is intentional and are performed by those who have a good understanding of the underlying system. Such attacks are popularly known as insider attacks.

Based on our literature review, we have identified four major questions that can guide towards better handling of insider attacks: (a) who can perform these attacks? (b) what gets affected? (c) how to detect these attacks? and (d) how to prevent them from happening? Figure 2.1 gives a list of entities to consider when dealing with insider attacks. The figure also shows the four questions, from above, as relationships among the entities. Insider attacks can be performed by (a) traitors who are legally a part of the system but want to misuse the access privileges given to them; (b) masqueraders who get access to the system by stealing identities of those who have legitimate access. Insider attacks can affect the proper functionality of a program or corrupt the data used by the programs. Profiling and trapping are two common ways to detect insider attacks [17, 25]. Profiling can be performed (a) at the program level [36, 37] and at the user level [38]. Techniques for profiling user behavior and network behavior have evolved over time and the latest techniques use machine learning algorithms to achieve this [39, 40]. User profiling is performed by observing resource utilization [41] and psychometric analysis [42]. Traps can be set in the programs

17
or in the network to force the attacker into performing certain actions that help towards exposing the attack [43, 44]. The biggest concern with these insider attack detection methods is the possibility of losing valuable data. Hence, insider attack prevention mechanisms such as identity management [45, 46], access control lists [47, 18], data encryption [48, 49] etc must be employed at the same time.

Insider attacks are a dangerous security problem in any domain because they are difficult to predict and detect [25]. Hence organizations must try to safeguard their systems and data from insider attacks [50]. Predictive models for user/program/network behavior with the help of continuous monitoring is a widely adopted solution for insider attack detection. But such prediction is not completely reliable and the difficulty in detecting attacks grows with the complexity of the underlying system. Recent advancements in computing led to wide adoption of services such as
cloud computing and big data which are extremely complex in their design and development. In cloud computing, many insider attacks can be performed by misleading the client side services and once compromised, data obtained can provide social engineering opportunities for cascade attacks [51]. Having security as a service model for cloud environments [52] and having sealed clouds [53] are some ideas proposed towards protecting cloud infrastructures from insider attacks. While cloud computing is more about computing on the fly, big data deals with organizing and managing large sets of data. Insider attack detection and prevention for big data frameworks is an area that is not well explored yet.

Insider attacks are known to be difficult to detect and prevent in general. This problem intensifies when the system under consideration is deployed on a large, distributed cluster. The ideal solution to detect and/or prevent insider attacks is by automating every aspect of a system such that there is no human intervention at all but obviously this is not feasible. Especially for big data systems, there is usually a service stack at the provider’s end and another service stack at the client’s end. Hence, cloud service providers such as Amazon and Google reduce the scope for insiders by adopting a two step procedure: (a) making most aspects of their systems to run automatically, and (b) asking their clients to do the same.

2.2.2 Programmer Errors

Programmer errors are a huge concern to security architects because anticipating the vulnerabilities due to programmer errors is difficult but at the same time they can give leeway to the attackers in their attempt to compromise a system. Programming errors often lead to insecure interaction between system components and might create exploitable vulnerabilities in the applications. A popular threat caused due to programming error is when an attacker obtains root privileges. Such
exploits let the attacker take advantage of buffer overflows, race conditions, access to system files that should be restricted etc. But such vulnerabilities can be handled by continuously auditing the programs and having restricted access rights [54]. This is a technique that is widely adopted by the industry in the recent past by leveraging big data platforms. In other cases, programmer errors allow an attacker to misuse memory and data. Generally, such vulnerabilities can be mitigated by enforcing control-flow integrity [55]. More specifically, programmer errors that lead to memory corruptions can be handled by sanitizing memory instructions [20] in a program at compile-time. Though this approach is memory intensive, it seems to work efficiently for applications that run on a single machine. Using sanitizers in real-time distributed applications is not feasible. In this work, the concentration is only on a subset of programmer errors that corrupt memory and cannot be detected until runtime.

### 2.3 Security in Distributed Systems

Security in distributed systems has been well researched over the past two decades. For this work, the focus is in two aspects of security in distributed systems: intrusion detection and tracing system behavior to identify or predict such intrusions.

#### 2.3.1 Intrusion Detection

Intrusion detection systems (IDS) are used to detect anomalous or malicious usage in a computing device. Common design strategies for IDS are based on: (a) knowledge from prior attacks, and (b) learning from the behavior of programs and/or users. Knowledge-based IDS usually searches a program for known threat signatures that are stored in a database. With the drastic increase in the number of zero-day attacks, relying on a pre-populated database of threats is unsuit-
able. Even if it is assumed that an ideal database of all possible threats exists, maintaining it would require a lot of resources and running search queries against it would be expensive. On the other hand, behavior-based IDS tries to model, analyze and compare user and/or application behavior to identify anomalies. Network usage (packets of data) is another common component observed by such IDS. This technique needs more resources and is more complex than signature-based IDS but it is more effective in a dynamically changing threat environment. Behavior-based IDS generally try to capture the context and apply statistics and rules on that context to detect anomalies.

A distributed implementation of IDS is needed for systems that run on large clusters. Such an IDS would have centralized control and can detect behavioral patterns even in large networks. Efficient ways of data aggregation, communication and cooperation are key factors of success for such distributed IDS and it has to be employed at multiple levels: host, network and data [56]. Hence, using big data platforms to support general-purpose distributed IDS implementations is a recommended and popular practice. Some of the popular IDS using big data are: Apache Metron [57], Apache Spot [58], IBM Guardium [59] etc. But in this work, the aim is to build an IDS that can be used for security within a big data platform itself. IDS within a big data platform favors behavior-based distributed IDS because of the naturally large and ever increasing scope of threats.

2.3.2 Tracing

The need for tools that can diagnose complex, distributed systems is high because the root cause of a problem can be associated to multiple events/components of the system. Recent works in the distributed tracing domain are concentrating on providing an independent service. Magpie [60] works by capturing events in the distributed system and uses a model-based system to store the traces. Xtrace [61] provides a comprehensive view for systems by reconstructing service behavior
with the help of metadata propagation. Though it has similarities to the proposed approach of providing task-centric causality, Xtrace concentrates on network level analysis. Retro [62] is another end-to-end tracing tool that audits resource usage along execution path. The drawback with tools such as Xtrace and Retro is that they are tightly coupled into the system and hence need the user to modify source code. HTrace [63] is an Apache incubator project for distributed tracing which requires adding some instrumentation to your application. Pivot Trace [64] is a dynamic causal monitoring service for distributed systems that provides a happened-before relation among discrete events. Fay [65] is another distributed event tracing platform that instruments code at runtime with safe extensions. G² [66] is a graph processing system for diagnosing distributed systems. Finally, [67] proposed an anomaly detection method for MapReduce jobs in Hadoop by collecting provenance data.

In this work, the idea is to trace system & library calls. Detecting intrusions using system call stack has been explored before. Hofmeyr et.al [68] show that short sequences of system calls executed by running processes are a good discriminator between normal and abnormal operating characteristics. Many models such as Bayesian Classification, Hidden Markov Models (HMM) and process algebra have been proposed for system call sequence analysis [69, 70, 71, 72]. In this work, system and library call metadata is used to build behavior profile of a process. This is done by extracting information about system calls made during runtime from the call stack. Also, information related to library calls is included in the behavior profiles because big data frameworks use library calls that can be completely accounted for. This aspect of the proposed approach is similar to AWS CloudTrail [73] which enables user to retrieve a history of API calls and other events for all of the regions within the user’s account.
2.4 Security in Big Data

Security in big data is gaining tremendous momentum in both research and industry. But big data security is overwhelmingly inclined towards leveraging big data’s potential in providing security for other systems [74]. Security within big data systems is still a budding phenomenon. It is ideal to include security as a major component in the holistic view of big data systems. But the requirements of big data applications such as real-time data processing, fault tolerance and continuous availability give little scope to employ complex and robust security mechanisms. All existing security techniques implemented within big data frameworks are software based and try to prevent external entities from attacking the system.

For example, Hadoop security model is built on three pillars: multilevel authentication, log-based analysis and encryption. Strong authentication is provided to Hadoop by Kerberos [18] which is an integral part of the Hadoop ecosystem. Advantages of having such simple software oriented security mechanisms, such as Kerberos, are better performance and simple management. The main requirements in Hadoop security design focus only on access control [75]. But there are various problems with such a policy enforcing security software, as identified in [76] and [77]. Also, none of these approaches can strongly counter insider attacks. Newer versions of Hadoop have an option to configure the Hadoop cluster in secure mode which supports multiple levels of authentication. Activity at any level inside a Hadoop system is logged with the help of in-built services such as log4j [78]. These logs are extensively used by the system level services and administrators for various management purposes such as understanding resource utilization and security monitoring. Encryption is an option available in the recent versions of Hadoop to support data confidentiality between services and clients. Other software services such as Zookeeper [79] and research projects
such as Rhino [80] and several other open jiras are continuously working on improving Hadoop security. But some problems with existing Hadoop security services are:

- **Having centralized control.** Kerberos, Bastion boxes, Zookeeper etc need synchronization and incur delay in risk assessment and recovery.

- **Being integral to the system.** All security services are built into the Hadoop ecosystem and hence need to be implicitly trusted.

- **Burden on the main processor.** Even the optimized encryption/decryption standards such as AES-NI[81] need minimum 2 real cpu cores for efficiency.

- **Come with unreal assumptions.** According to any Hadoop admin manual, users cannot have access to machines in the cluster that store their data.

According to Hadoop Security Design[75], permissible performance overhead for a change in architecture is only 3%. This is precisely the reason behind coarse-grained security mechanisms such as data encryption being an optional and restricted feature in big data systems. Data encryption in Hadoop is only available for data that gets exchanged between user and the system but not for data that travels within the system. Randomized data encryption for data security was proposed in [82] but this work acknowledges that faster results are yet to be achieved. Attack tolerance for big data systems through redundant fragmentation was proposed in [83] where the idea is to stripe data chunks and re-assemble them. This idea is based on redundancy and does not show any experimental results about the overhead involved. Also, big data properties such as large scale distributed infrastructures and replication make it difficult to detect insider attacks precisely using the traditional methods.
In this work, the inefficiency of existing big data security mechanisms is demonstrated by implementing two insider attacks on a big data cluster. The four attack scenarios implemented and demonstrated in this work are (a) an insider from Hadoop ops team who has access to a machine/log file can modify or delete it, leading to data loss (b) an insider from system admin team can tamper with log data which leads to faulty results and data degradation (c) an insider who has access to configuration files can modify datanode configuration that can result in bad system performance, and (d) an insider who has access to user data can distribute it and impact differential privacy.

2.4.1 Attack 1: Data Loss

Metadata about the cluster file system is stored on Namenode in 2 parts: \textit{fsImage} that is used when the HDFS services are started and \textit{EditLog} that is updated constantly (hdfs-default.xml or hdfs-site.xml has the configuration settings related to this). As per the initial Hadoop architecture, Namenode was a single point of failure. But using a secondary namenode to store HDFS information resolves this problem in most cases. Also, the secondary namenode helps in making the file system updates invisible to the user. This is done by having periodic updates to the \textit{fsImage} on secondary namenode with the help of local \textit{EditLog} and dumping the resultant \textit{fsImage} to namenode in a snapshot. In this scenario, if an attacker (insider from Hadoop ops) modifies the \textit{EditLog} on secondary namenode, the next checkpoint will reflect that change on the \textit{fsImage} of namenode which can lead to data loss. The same attack scenario was implemented on the Hadoop cluster. A high level model of the attack is given in figure 2.2a and the results of the attack are given in 2.2b. In figure 2.2a, Op represents an insider from the Hadoop ops team. From figure 2.2b, it can be noticed that the file system originally had 297 data blocks. But after the \textit{EditLog} on secondary namenode got modified by a script, the existing file system i.e. \textit{fsImage} got erased completely. This
change was reflected in the namenode at the next checkpoint i.e. after 10 minutes. A system admin who has access to the secondary namenode in a Hadoop cluster can implement this attack. This is an example of loophole in the Hadoop architecture that can be misused easily by insiders.

2.4.2 Attack 2: Data Degradation

Hadoop logs are mostly used for management purposes. From a security standpoint, server logs are used by system admins to identify potential attacks on the system. Log data can be stored in structured formats such as tables using Hadoop scripting languages such as Pig and Hive. System admin can then use BI tools such as MS Excel to easily analyze the structured log data. Flume and Kafka are two popular big data products for real-time event processing. Most big data analysis and security solutions tend to use these services within their framework. An example of system log monitoring is given in a tutorial by Hortonworks [84] where DDOS attacks are being tracked down by system admins. As per the workflow in this example, users requests the client service to access data stored in HDFS. These user requests will all be logged by the log4j service. Hence, any attackers requests will also be logged. The system admin can easily build a framework with the help of services such as Flume, Hive and Hcatalog to monitor and track the user requests. In this example, Hortonworks used Flume for streaming data transfer of Hadoop logs into Hcatalog which is a SQL based Hadoop data store. Interestingly, this tutorial can be used as a counterexample to show that admin can act as a traitor, manipulate the server log data and create results that depict a wrong picture to the higher administration. In this example, an attacker (insider from system admin) can alter the results by modifying the log data before Flume can stream it into Hcatalog. A model of this attack scenario is shown in figure 2.3a where 1,2,3 are examples of DDOS attack requests from client side that will be logged by the system. A malicious script from the
(a) Attack Model to Create Data Loss.

(b) Example Showing Actual HDFS Data Size.

(c) Example Showing Data Loss in HDFS after Attack.

Figure 2.2: A Scenario for Data Loss in a Hadoop Cluster.
Hadoop ops side (as an insider job) is run on the system that would modify log data before Flume starts streaming it. From figure 2.3b it can be noticed that it is possible to taint the results by implementing attacks on log files even though Hadoop services seem to be working as expected. The part of figure 2.3b labeled Before shows the actual data of origins of user requests and attacks while the part labeled After shows the modified output.

### 2.4.3 Attack 3: Data Exposure

This attack involves two cases: (a) the use of non-certified (and untrusted) equipment to transfer data from one machine to another, and (b) the use of certified and trusted software (such as a mail client) to transfer data from one machine to another. Similar to the previous attack, the first step involved in this attack is for the system admin to modify the configuration through the `hdfs-site.xml` file on of the datanodes of the Hadoop cluster. A new location local to the system admin account is added to the DFS data directory property. As a result, all blocks at this datanode have two copies - one copy in the actual HDFS location used while setting up the cluster and another duplicate copy in the system admin’s local folder. Next, a script is used to simulate an insider periodically transferring these duplicates files from his local folder of to another remote location using the mail client service or USB device. Since it is not possible for us to connect a USB device to Amazon EC2 instances, we found the system calls involved when using such a device and called them in the attack script.

### 2.4.4 Attack 4: Infrastructure Degradation

This attack involves exploitation of access privileges by an insider who is legally allowed to access the system and its configuration files. An insider who is a system admin can modify
(a) Attack Model for Faulty Results

(b) Example Showing Actual Analysis Results.

(c) Example Showing Manipulated Analysis Results.

Figure 2.3: A Scenario for Data Degradation in a Hadoop Cluster.
the configuration properties of a datanode to intentionally impact the performance of the overall system. To implement this attack, the system admin changes the datanode configuration through the `hdfs-site.xml` file on one of the datanodes of the Hadoop cluster. The amount of memory allocated for non-DFS purposes on the datanode were increased by 25% and the number of server threads for the datanode were reduced by changing the handler count to 2. Since this is a one-time modification made by an authorized user whose job entails modification of the configuration files, usual user-profiling will not help in detecting the attack.

### 2.5 Special Hardware for Security

Hardware based security gained popularity in the last decade. Trusted Platform Module (TPM) a device specification that enforces transitive trust in computing platforms by providing methods for collecting and reporting the hardware and software components of a system. Versions of TPM such as vTPM have also been proposed which can extend TPM to unlimited number of virtual machines on a single hardware platform [85]. TrInc [86] is another security hardware based on non-decreasing counters and shared symmetric key encryption. Though this approach eliminates the need for trusted hardware, it is only applicable to equivocation-type misbehavior and is not scalable. Ascend [87] is a data privacy conserving scheme for cloud such as environments. It obfuscates instructions of a program to implement data privacy. But this comes at a cost of modifying the ISA, average slowdown of 10-15% and no proper limits on obfuscation. Raksha [88] proposes to use a dedicated tag coprocessor and memory for dynamic instruction flow tracking. With a slowdown of less than 1% these coprocessors need to be synchronized with the main processor core and are confined to individual machine level security.
Proposing hardware oriented security methods for Hadoop are on the rise in recent times. A TPM based authentication protocol for Hadoop was proposed by [89] which claims to be much faster than Kerberos, though it has not been fully implemented. A hardware oriented security method to create trusted Apache Hadoop Distributed File System (HDFS) was proposed in [90] which is a theoretically novel concept but was proven to work only on one node. The overhead of data encryption by TPM acts as a hindrance in adopting this method, especially when the size of data maintained in big data systems is ever growing. In this work, the idea of delegating security in big data systems takes center stage. This is realized by designing an independent security framework with the necessary components that can be hosted on secure hardware. The components of such an architecture include security techniques capable of analyzing processes at both compile-time and run-time. Though a set of five different security techniques are proposed in this dissertation, the generic nature of the framework enables it to host other techniques in the future as well.

2.6 Control Flow Graphs

A control-flow graph (CFG) is a directed graph representation of a program and usually a sparse graph. CFGs include all possible control paths in a program. This makes CFG a great tool to obtain control-flow behavior of its process. Vertices in a CFG give the level of detail, such as instruction-level or basic block level, that cannot be further divided. Edges in CFG represent control jumps and are classified into two types - forward and backward. Branch instructions, function calls, conditional and unconditional jumps account for forward edges. Virtual calls and indirect function calls are also considered as forward edges but their destinations are difficult to determine. Loops and returns generally account for backward edges. The integrity among duplicate processes that run on replica nodes of a big data system can be verified with the information available in a CFG.
Similarity check between program logic of two programs can be performed by comparing their CFGs for isomorphism. There are many ways to check for such graph isomorphism [92, 93] but analyzing the similarity of two processes by conducting CFG level graph isomorphism is hard and time consuming. Graph isomorphism is a complex problem, sometimes known to be NP-complete as well [94]. To reduce the complexity of graph algorithms, CFGs can be reduced to trees or subgraphs before performing any coherence or integrity checks [95]. A CFG can be converted to a tree using methods such as Depth-first traversal. Several tree structures such as Dominator Tree, Minimum Spanning Tree (MST), Minimum Spanning Arborescence (MSA) can be extracted from CFGs [96, 97, 98]. CFGs can be broken into subgraphs using methods such as k sub-graph matching and graph coloring. Assuming that a CFG has \( n \) vertices and \( m \) edges, some popular methods for graph reduction and graph comparison that can be found in the literature are:

- **Based on Edit Distance**: Using Smith-Waterman algorithm with Levenshtein distance to identify similarity between two graphs that are represented as strings [99]. The time complexity is \( O(nm) \).

- **Based on Traversal**: (a) A preorder traversal of a graph \( G \) where each node is processed before its descendants. (b) A reverse postorder in a DAG gives a topological order of the nodes [100].

- **Based on Dominator trees**: Using a tree data structure built using the method proposed by Tarjan in [101]. This method has a time complexity of \( O((n+m)\log(n+m)) \). Depth First Search can also be used.

- **Based on Reachability**: Applying transitive reduction of a sparse graph to convert CFG to another graph with fewer edges but same transitive closure [102]. The time complexity is \( O(nm) \).
In this work, a CFG is reduced to a set of MSAs (directed MSTs) because CFGs are generally sparse graphs and hence the size of the set of MSAs is finite and usually small in number (less than 100). In such contexts, Edmond’s algorithm can be used to quickly extract all MSAs from a digraph [96, 97, 98]. Since an MSA contains all vertices of its graph, there will be no loss in the program instruction data. Depending on the connectedness of the graph, the edge count will defer between the CFG and MSA representation of a program. Figure 2.4 shows transformation of a line of Java code to basic blocks of bytecode to CFG to set of MSAs. Vertices B1, B2, B3, B4 are the basic blocks formed from Java bytecode. There exists an \(O(m + n \log n)\) time algorithm to compute a min-cost arborescence [96]. Alternately, another approach for converting a CFG to MSA using union find is used by popular compilers such as llvm and gcc for security purposes [103]. One known disadvantage of using CFGs and MSAs for security is that dynamic link library calls cannot be verified.
Traditionally, IDS checks for known malware in programs by performing signature matching on a threat database [104]. Signature match using exact string matching is limited in its scope. This is because variants of same attack will have different signatures. Recently, methods to detect new malwares using statistical machine learning have been proposed. Static analysis using CFG is another efficient way to detect intrusions but it is very complex [105]. Converting a CFG to a string and implementing string matching is another way to deal with this problem but the solution will not be polynomial. Also, CFG at basic block level can have basic block variants that look different but perform the same function. To deal with these shortcomings, many approximate matching techniques have been proposed. Tracing applications to get their CFG is another approach that is used in applications such as xtrace, pivottrace etc [106, 107]. In case of big data systems, data nodes usually have the same processor architecture. Hence it can be assumed that there will be no variants when the CFG is constructed at byte-level. It is then sufficient to verify similarity among the CFGs of two processes to confirm coherence in the nodes of a big data system.

Control Flow Integrity (CFI) [108, 109] which is another popular and effective technique for attack prevention which enforces the execution of a program to follow a path that belongs to the program’s control flow graph. The set of possible paths are determined ahead of time using static CFG [108, 109]. A coarse-grained or fine-grained version of CFI can be used for program profiling. But the problem with any such profiling techniques is the overhead incurred in conducting them, even more if performed remotely. Though such limitations of this approach have been identified [110], it is accepted as a strong and stable security enforcing mechanism. There are a plethora of CFG-based code similarity algorithms [111]. But such CFG similarity check methods are complex, expensive, have no defined standards. Most CFG similarity algorithms rely on some simplification
techniques such as fingerprints, edit distance, comparison only with known graphs in a database etc. Also, the impact of CFG similarity analysis differs a lot depending on when and how the CFG is generated for a program. These complexities and uncertainties led to a new set of control flow analysis techniques that avoid translating the program code to a formal model. For example, insider attack detection based on symbolic execution and model-checking of assembly code was proposed in [112]. In this work, a novel approach for control flow similarity check for attack detection is proposed that completely discards the idea of building CFGs. Instead, the proposed idea is based on simple string matching of control instruction sequences obtained from assembly code of scheduled processes. Another software testing technique that was proposed many decades ago and resurfaced again recently is called symbolic execution [113]. This technique suffers with path explosion as well and hence it is usually handled with the help of search heuristics [].

2.7 Memory Access Patterns

Understanding memory access patterns of big data applications will help in profiling them from their data usage perspective. Patterns in bandwidth usage, read/write ratio or temporal and spatial locality can be used when observing memory accesses of a process. For example, W.Wei et al. [114] observed that memory access patterns of big data workloads are similar to traditional parallel workloads in many ways but tend to have weak temporal and spatial locality. One of the first works in characterizing memory behavior of big data workloads was done by Dimitrov et al. [115] who observed the characteristics such as memory footprints, CPI, bandwidth etc. of the big data workloads to understand the impact of optimization techniques such as pre-fetching and caching. Other works such as [116, 117] explored the memory behavior of big data workloads at cache and virtual memory level. In distributed compute systems, nodes of the cluster are typically
virtual machines. For example, a Hadoop datanode is a process which dynamically dispatches tasks every time a job is scheduled. So, profiling the sizes of the private and shared memory accesses of all tasks will give the memory access pattern of the datanode.

### 2.7.1 Principal Component Analysis

Principal Component Analysis (PCA) is an unsupervised linear transformation technique that finds the directions of maximal variance in a given dataset [118, 119, 120]. A principal component is a linear combination of all the variables that retains maximal amount of information about the variables. When used for fitting a linear regression, PCA minimizes the perpendicular distances from the data to the fitted model. This is the linear case of what is known as orthogonal regression or total least squares [118], and is appropriate when there is no natural distinction between predictor and response variables. This is perfect for memory access pattern matching problem because the features of memory access are all random and do not follow any predictor-response relation. Of course, PCA is only one of the tools to achieve the desired results for this problem. Other techniques like Guassian Mixture Models (GMM), tests for homogeneity of variance such as Box’s M test or Bartlett’s test can also be used.

According to the theory of orthogonal regression fitting with PCA [118], $p$ observed variables can fit an $r$ dimensional hyperplane in $p$ dimensional space where $r$ is less than equal to $p$. The choice of $r$ is equivalent to choosing the number of components to retain in PCA. For this work, $r$ and $p$ are the same because reducing the dimensionality of the dataset is not needed. But to profile memory usage of a process and then compare it with other profiles, a function that can explain the memory behavior is needed. For this purpose, T-squared values are used that can be calculated using PCA in the full space.
2.7.2 Long Short-term Memory

Neural networks are efficient tools to build predictive models that can learn regularities in simple sequences [121, 122]. A neural network is usually a feed-forward network built using perceptrons that take multiple inputs and use an activation function, some weights and biases to compute a single output. It learns to adjust the weights and biases over time using an optimization function called gradient descent algorithm. Back propagation [123] is the most popular method for gradient calculation that gives an insight to how quickly the cost changes when the weights and biases are changed in a neural network. Feed-forward neural networks are the most commonly used versions of neural networks and they are used to solve classification problems. In recent times, a modification of neural networks called recurrent neural networks (RNN) is gaining popularity because it uses a chain-like structure and can connect the perceptrons in a directed cycle that creates internal memory to store parts of the input sequence and intermediate results [124]. But back propagation in such networks takes extremely long time. Also, standard RNNs have to deal with the vanishing or exploding gradient problem where they eventually fail to learn in the presence of time lags between relevant input events and target signals [125]. To solve this problem a gradient method called Long Short-Term Memory (LSTM) was introduced that can help preserve the error that can be backpropagated through time and layers [126].

LSTM networks work by using cells that are made up of a set of gates: forget gate (FG), input gate (IG), cell gate (CG) and output gate (OG). These cells and their gates help in retaining, adding or deleting information from one pass to another. A vanilla LSTM implementation use all gates and relies on activation functions such as sigmoid ($\sigma$), tanh for proportional selection of data. While sigmoid ($\sigma$) function is used to open/close a gate in the LSTM cell, the tanh function is
applied to cell state and output selection where the result can be positive/negative. LSTM also uses stochastic optimization functions such as Adam [127], RMSprop [128] or stochastic gradient descent for measuring the gradient in the output. Since there is no conclusive evidence for which optimization function works best for LSTM, Adam is randomly chosen for this work. Figure 2.5 and Equations 2.1 give a basic overview of how a LSTM cell works. Here \( w_{fg}, w_{ig}, w_{og}, w_{cg} \) are weights and \( b_{fg}, b_{ig}, b_{og}, b_{cg} \) are bias associated to gates while \( t-1, t \) are two consecutive steps in a sequence.

Every LSTM cell has three data values at each step: cell state (\( c \)), input (\( i \)) and output (\( o \)). Many variations have been proposed to vanilla LSTM [129] but researching all such variations is not the main focus for this work.

### 2.8 Context of this Research

The current state of the art for security in general is driven by the more popular security risks such as injection, cross site scripting and broken authentication. In this work, the problem of insider attacks is investigated thoroughly. The target environment for this research is a big data system and the focus is on protecting sensitive data from exposure and destruction. The current state of the art for security in big data systems is access control, log monitoring, data encryption (at rest and in transit) and network analysis. These methods protect data from attacks that originate...
outside the system and the attacker is an external user. A set of techniques are proposed to detect and predict insider attacks and significantly advance the current state of the art.
CHAPTER 3 : PROPOSED FRAMEWORK

The main strategy proposed in this dissertation is to delegate security of big data frameworks that run on distributed compute environments to secure hardware. To realize this, a set of security techniques are proposed that exploit big data system properties such as replication. However, an efficient framework is needed to develop a complete security solution that can tie these techniques together. Hence, a security framework is also proposed that is built with components that integrate to support the two step security solution: (step 1) analyze programs locally and, (step 2) verify analysis results dynamically.

3.1 Local Analysis

The first step in the holistic view of the proposed security solution is to analyze programs locally at slave nodes. Each slave node i.e. datanode can perform such local analysis on two separate occasions. The first stage of analysis is performed before the program is scheduled to run. This step is realized using one of the proposed compile-time security techniques. Primary focus of this static analysis is to detect security holes in a program by verifying the control-flow obtained from program binaries and using instruction level rules and heuristics. The next stage of analysis is performed during execution of a program to detect and predict attacks. The final analysis is performed after the program terminates. Analysis that is done during or after program execution use the run-time security techniques. This stage is also known as the Process Profiling stage. In this work, run-time analysis is limited is call and memory usage of a process.
3.2 Dynamic Verification

Besides performing local analysis of programs that run on them, each datanode also verifies program information it receives from other replica datanodes. This verification step is a matching technique where local analysis results are compared with received analysis information. Methods and heuristics used for this matching process depend upon the features examined. Some examples techniques that can be used in the matching process are: graph and string matching techniques, statistical analysis tests and deep learning techniques. After the matching process, datanodes share their analysis results with the replica datanodes and reach a consensus about the existence of an attack. Consensus algorithms like raft [130] and paxos [131] or any simple home-brewed consensus solution can be used in this stage for communication among replica datanodes. This stage is also known as the Matching stage or the Verification and Consensus stage. Figure 3.1 shows a high-level overview of the proposed security architecture.

3.3 Secure Communication

A big data system is technically a distributed data storage system that relies on secure and efficient communication protocols for data transfer. The proposed system aims to provide robust security for big data systems by having a modular design and being independent from the core big data services. For this reason, a separate secure communication protocol is included in the proposed framework that can be isolated from the set of default communication protocols used by the big data system.

The proposed framework is a mix of independent security modules that work together and reside on individual nodes of the system. These modules need a secure communication protocol to
share packets of data with their counterparts on other nodes of the cluster. The data shared among the security modules contain vital information about the analysis of a process. Hence, a public key cryptosystem based on RSA key exchange [132] is used in the proposed secure communication protocol. Simply put, the secure communication protocol uses asymmetrical public and private keys to encrypt and decrypt data. Pairs of keys are created together where each key is a large random number generated from a set of hardcoded keys that are not accessible to anyone. All data transferred by any node using this secure communication channel is encrypted upfront using private key encryption. The associated public key will be shared with all other replica nodes that a data node has to communicate with. This simple idea can be realized with the help of hardware security chips such as TPM [29] or Intel’s TXT [30] that have public-private key encryption modules. Such hardware security chips come with a hardcoded, on-chip master key. A simple random number
generator module is used to generate public-private key pairs periodically using the hardwired master key.

In this work, SSH protocol and RSA encryption algorithm for key exchange are used together for secure communication though any such cryptosystem will work. Given the off chance of leakage of private keys, a key pair is held active for only a certain time period $T$. This increases the robustness of the communication protocol. In this work, the focus is not on finding the perfect value for $T$ and hence a good value for $T$ is assumed to be 10 seconds. The public key of a node is shared with all other nodes it has to communicate with i.e. replica nodes and master node. All incoming data packets to a node will be encrypted with its current public key and can only be decrypted using the corresponding private key that is stored locally. Once the messages are decrypted, information will be sent to the process matching module to identify attacks using various similarity measuring techniques.

Given the short lifespan of public keys used in the secure communication protocol, each node should be able to store public keys of all other nodes it has to communicate with. Also, storing older keys of other nodes helps in verifying authenticity of nodes in case of attack recovery. Hence, a queue data structure is used on every node to store the periodically generated public keys of other nodes. Back of $\text{queue}_n$ will be the latest public key to be used for encrypting packets to be sent to node $n$ while front of $\text{queue}_n$ will be deleted when $\text{queue}_n$ is full (to accommodate a new key). Limiting the maximum queue size by some $k$ will make sure that a node has enough information to support attack recovery measures while not consuming too much memory. Finding the perfect value for $k$ is outside the scope of this work and hence a predefined value of 5 was used while conducting the experiments.
Algorithm 1 shows the steps involved in the proposed secure communication protocol. Once a model of the proposed system is installed, all nodes will periodically generate public-private key pairs for as long as the system is in use. This is accomplished with the help of the hardwired key on the special purpose security chip and the random number generator module. At the end of every $T$ time units, a new public-private key ($newkp_n$) is generated on a node for communicating with replica node $n$. The private key $priv_n$ of $newkp_n$ will be used for decrypting incoming data from node $n$ and the public key $pub_n$ of $newkp_n$ will be shared with node $n$. For ease of access to keys during decryption, current private keys of all nodes are stored in an array $arr_{priv}[]$. The loop given between lines 4 and 9 in the algorithm 1 depict the key generation phase. Once a public key $pub_n$ is shared with node $n$, all incoming messages from node $n$ will only be decrypted using the associated $priv_n$ for the next $T$ time units. An array of queues, $arr_{pub}[]$, is used to store public keys received from all other nodes. When a node has to send an message $msg$ to replica nodes, the public key of that node is used to create an encrypted message $msg_e$. The loop given between lines 10 and 14 in the algorithm 1 depict the key usage phase for encryption and decryption of messages shared among replica datanodes.

3.4 Model of the System Architecture

The proposed framework is a combination of 3 parts: local analysis, dynamic verification and secure communication protocol. As shown in Figure 3.1, these three parts are made of multiple modules that need to be installed on all nodes in the big data system. Also, locality of these modules impacts the performance of the system greatly. The closer they are to the main processor of a node, the faster and less expensive it will be to communicate. However, from a security standpoint these modules need to be isolated from the big data system main workflow. Hence, the model for the
Algorithm 1 Secure Communication Protocol

1: procedure Secure Communication Protocol 
2: while true do
3:     if time = T then
4:         for n in OtherNodes do
5:             newkp_n ← get new public private key pair from TPM chip 
6:             pub_n ← get public key from newkp_n 
7:             priv_n ← get private key from newkp_n 
8:             node_n ← send(pub_n) 
9:             arr_priv[n] ← priv_n 
10:         for n in OtherNodes do
11:             if queue_n = full then 
12:                 dequeue(queue_n) 
13:                 queue_n ← enqueue(pub_n) 
14:                 arr_pub[n] ← queue_n 
15:         msg ← to be sent to all replicas 
16:         for r in Replicas do
17:             pub_r ← back(arr_pub[n]) 
18:             msg_e ← encrypt(msg, pub_r) 
19:             send(msg_e) 

The proposed system is designed to fit on isolated special purpose security hardware chips. Such chips can be built on top of existing security hardware such as TPM or Intel’s TXT chips [29, 30]. When compared to a software solution that can be very adaptive, hardware solutions are popularly known to affect the scalability and flexibility of the big data infrastructure. Such problems are avoided in this work by decoupling the proposed solution from the actual workflow of the big data platform.

There will be a one-time extra cost due to the hardware security modules. An overview of the elements in such a model of the proposed system is given in Figure 3.2 where modules with bold boxes represent the proposed security techniques. The functionality of each of these elements is as follows:

- Analyzer, this module will get the data from the hotspot VM and perform the initial steps of cleaning the data. Result from analyzer is stored in Memory.
• CFI filter, this module takes input, a set of assembly language instructions, from the Analyzer module (technically, the Memory module) and filters out the control flow instructions, while maintaining the order.

• Statistical Tester, this module implements the various statistical tests to be performed on memory information such as PCA, ANOVA, Tukey etc.

• RNN, this module implements a Recurrent Neural Network (RNN) for attack prediction from memory information.

• APS Scorer, this module calculates the Attack Probability Scores (APS) of programs with the help of rules defined in the Look Up Table (LUT).

• Sequencers, there are three sequencers in the proposed model, one each for jumps, calls and...
returns. Each sequencer goes through the output of CFI filter module and forms a delimited sequence string of the instruction it is associated with. Then, the sequencer uses the SHA hasher module to generate and store a fixed length hash output from the variable length instruction sequence string.

- Register Array, there are 4 registers in this array to store message, jump instruction hash, call instruction hash and return instruction hash.

- Message Register, this is a special register in the Register Array used to store the message in thread-safe manner.

- Message Generator, this module combines all the individual hash outputs stored in registers and uses the SHA hasher module to generate a fixed length hash output. This hash of hashes is combined with the process metadata to generate and store a message that represents the process.

- Encryptor / Decryptor, this module uses the Key Store to access the current set of public/private keys and the Message Register to access the current process message. The Encryptor module uses the public key of a replica node from the Key Store and encrypts the message in Message Register. The decryptor module uses the private key of the node from the Key Store to decrypt an incoming message.

- Comparator, this module performs string comparison between local message (hash of hashes) and received message.

- Key Generator, this module uses the underlying TPM/TXT chip’s [29, 30] in-built functionality. The hardwired key and the random number generator of the security chip are used
to generate a new public/private key pair; and the timer of the chip to trigger this action periodically.

- Key Store, this module uses an array of memory locations to store the public key queues of all replica nodes and the current public / private key pair of this node. The three most recent public keys of each replica node is stored in its queue.

- Exchanger, this module uses TCP/IP protocol to exchange messages with other nodes.
CHAPTER 4 : COMPILE-TIME TECHNIQUES

The first set of program analysis techniques proposed in this work are static compile-time driven methods. These methods focus on extracting control-flow information from compiled programs that are scheduled to run on the big data cluster. Each technique uses a different strategy to achieve this, both at the local analysis step and the verification step. Assumptions and goals about (a) the surveyed environment, and (b) the attacks under consideration must be listed before exploring the details of the techniques. For this purpose, an attack model is given upfront.

4.1 Attack Model

This attack model focuses on two use cases: (a) misuse of log data, and (b) exploitation of programmer error vulnerabilities. As per the Common Vulnerability Scoring System (CVSS) the attack vector is local for these attacks because the exploitable scope is via read/write/execute [133]. The scope of possible insiders is confined to the system admins of a big data cluster. Also, as per hadoop security standards [75], system admins do not have access to the entire cluster. Security features such as data confidentiality and operational guarantees such as correctness of results can be compromised according to this attack model. The goals of an insider conducting such attacks can vary from personal vendetta to financial gain. The proposed system targets such specific insider attacks because they are relatively easy to implement with existing security solutions on platforms.

---

1 Portions of this chapter were previously published in Santosh Aditham and Nagarajan Ranganathan. A novel framework for mitigating insider attacks in big data systems. In Big Data (Big Data), 2015 IEEE International Conference on, pages 1876-1885. IEEE, 2015. Permission is included in Appendix A.
such as Hadoop and Spark. Attacks targeting misuse of log data can be performed by creating malicious programs or by modifying of existing program binaries with malicious intent. Given the existing security features of user-level activity monitoring, the possibility of system admins creating and executing new malicious programs is excluded from the scope of this attack model. Instead, the attack model focuses on system admins being able to modify binaries of existing programs. It is acknowledged that insider attacks are too broad and not all of them can be mitigated by the proposed solution. There can be other possible insider attacks in big data that are not visible at compile time and the proposed system may or may not be able to detect.

4.2 Technique Based on Attack Probability Score

The central theme to all proposed compile-time techniques is to reduce the amount of work needed to analyze a program binary. In this approach of detecting attacks based on Attack Probability Score (APS), a program binary is uniquely represented with the help of a numeric value (score) and two stack data structures.

4.2.1 Proposed Methodology

The main function of the proposed framework is to analyze processes to identify insider attacks. An attack in big data systems can be construed as unauthorized manipulation of data by a process running on one or more nodes of the cluster. As mentioned previously, coprocessors such as Rasksha [88] are capable of performing process level DIFT analysis with very low overhead. But according to [134] and [135], control flow instructions are more vulnerable to attacks than any other type of instructions in modern processors. Hence, it is safe to assume that the control flow of a process must indicate the possibility of an attack. Control flow of a process can be analyzed
before or after the process is scheduled. Analyzing a process after it is scheduled is beneficial for the framework and big data systems in general for three reasons: (a) easy to delegate to special hardware; (b) no impact on performance of the system, and (c) the fact that a program can be modified anytime by an insider before its object code is scheduled. Hence, process analysis in the proposed framework is limited to monitoring the control flow instructions from the disassembled object code (DOC) of a process that is generated using recursive traversal. So, a disassembler is used that can convert the scheduled object code (process) to assembly code. A disassembler that uses recursive traversal must be able to restore the control flow of the program in its output assembly code. Since the idea is to delegate such process analysis work to dedicated hardware, we take the liberty to assume safe and secure transfer of DOC of a process from main processor to such dedicated hardware.

A simple scan of the DOC for predefined control flow instructions enables us to mark the findings as possible areas of vulnerability in the code. But not all control flow instructions in the DOC can (or) will lead to an attack. Identifying an attack based on DOC level process analysis can be tricky for big data systems due to the possibility of false positives. An authorized administrator might be modifying the data as part of maintenance. Hence, a rule engine that categorizes instructions and data depending on their risk factor will be used while scanning the DOC of a process. Ideally, a rigorous rule engine is needed here that can differentiate attacks from authorized tasks. A full implementation of such rule engine is not the main focus and is out of scope for this work. Hence, a small rule engine is created with a small set of rules for few basic control flow instructions and data inputs. Table 4.1 gives an illustration of the rule engine used by this work. Such rule based process analysis is only a necessary but not a sufficient condition to confirm an attack in big data
systems. Confirmation of an attack can be achieved through consensus which needs communication among replicas. For such consensus, a node which suspects an attack on primary copy of data must be able to convey the same to its replicas using a simple message. Hence, a new metric called attack probability score (APS) is introduced.

APS is introduced in the proposed framework such that a node with primary copy of data can convey about a possible attack in a process to all the replica nodes. APS is important because it can alleviate the cost of comparing instruction and data stacks for every process at replica nodes. APS is based on % CFI of a process because it can: (a) capture the control flow property of a process to some extent (b) ensure same output for a process on any node (c) calculated easily, independently & internally by secure coprocessors and (d) used to sort processes in the compare phase of the proposed algorithm such that processes with high APS are compared first because they have higher chances to be attacked. APS value does not indicate if a process has been attacked or not. It is merely a process representation technique. APS is given at two levels: instruction and process. At instruction level, this score signifies attack probability of the instruction. Instructions and their APS are stored in a look-up table. APS values of individual instructions will be unique because they are derived based on the vulnerability quotient of the instruction. For example, as per general convention, a call or an indirect jump instruction is more vulnerable to attacks than an ADD instruction. At process level, APS has to represent a process based on process properties. Hence,

### Table 4.1: Basic Rule Engine.

<table>
<thead>
<tr>
<th>No.</th>
<th>Rule</th>
<th>APS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>call delete, call rename</td>
<td>very high</td>
</tr>
<tr>
<td>2</td>
<td>jump indirect</td>
<td>high</td>
</tr>
<tr>
<td>3</td>
<td>return</td>
<td>medium</td>
</tr>
<tr>
<td>4</td>
<td>all non-control flow instr.</td>
<td>low</td>
</tr>
<tr>
<td>5</td>
<td>all NOP instructions</td>
<td>very low</td>
</tr>
</tbody>
</table>
a logarithmic scoring rule is used, as given in equation 4.2 to calculate APS of process. Here, \( X \) is a boolean value to indicate if the process is attacked or not and \( P \) is the probability of having an attack. The probability \( P \) is calculated using equation 4.1 where a process has \( n \) instructions out of which \( k \) are CFI. During the compare stage a node can sort the processes to compare, depending on their APS value such that processes with more control flow instructions can be compared first. The basic steps to follow while calculating APS for a given DOC using equations 4.2 and 4.1 are given in figure 4.1. From the DOC of a program, the first step is to filter the control flow instructions (CFI). These CFI are pushed to the instruction stack in their order of appearance in the DOC. While doing so, the lookup table is referred to calculate the APS of that instruction. Next, the same steps are followed for the data field of the instruction. Here, the focus is more on the registers or memory locations used but not on the actual values of the data field. Thus, APS of an instruction is the sum of APS of its instruction code and data location. For all non-CFI, the APS value should be predefined to a value much lesser than its CFI counterparts.

\[
P = \frac{\sum_{cfi=0}^{k} APS_{cfi}}{\sum_{cfi=0}^{k} APS_{cfi} + \sum_{noncfi=0}^{n-k} APS_{noncfi}}
\]  

(4.1)
\[ APS_{\text{process}} = X \cdot \log(P) + (1 - X) \cdot \log(1 - P) \] (4.2)

Though APS is designed to represent the attack probability of a process, it does not uniquely identify a process. The mapping between APS and its source program is one to many. This is a problem for the attack verification step. Simple comparison of APS values of two programs is not sufficient to comment on the similarity of the two programs. A unique representation for every program is needed for this. Hence, the idea of representing a program uniquely with the help of a hierarchical model is proposed. First level of this model uses APS. The next level uses stack data structures where insertion and deletion follows a specific pattern. So, the combination of APS, instruction stack and data stack can uniquely represent a program.

Process analysis using DOC and an established rule engine can be very simple and straightforward with the help of appropriate data structures. A (key, value) pair format can be used to represent a control flow instruction and its APS. As mentioned before, identifying the general level of vulnerability of a control flow instruction and giving it an appropriate APS is an implementation detail that this work does not address. But assuming that it is taken care of, a lookup table containing all control flow instructions and their associated APS has to be maintained in static memory of the hardware that is performing the process analysis. This will help in faster access while retrieving APS of instructions and will not consume much memory. It is assumed that an instruction in DOC follows the format of opcode (arg1, arg2) where arguments arg1 and arg2 can be registers and/or constants. Two separate stack data structures: Instruction Stack and Data Stack are used to analyze DOC of a process. Having only push and pop operations makes stack a good fit to preserve the sequence of instructions and precisely identify the attack. This approach can be
associated to phase analysis [136] where the way a program traverses the code during execution is captured and analyzed. Also, while confirming an attack a complete scan of non-identical stacks is not needed which leads to faster analysis. Finally, having different stacks for instruction and data of each process makes data transfer, encryption and decryption more modular. The requirements of such hardware would be:

- Memory:
  - Stack data structures to store instructions and data values of a process under suspicion.
  - A static map to store all possible control flow instructions of the underlying ISA along with their associated attack probability scores.
  - Registers to store the hardwired on-chip key that serves as a master key and other intermediate keys used for encryption.

- Processing:
  - Hardware capable of running encryption algorithms to create packets for secure communication among replicas.
  - Hardware capable of performing analysis of Disassembled Object Code (DOC), calculation of APS and secure communication with replicas.

This algorithm runs inside the dedicated hardware as soon as the DOC of a program is received. The proposed framework is limited to checking the control flow instructions but other factors such as data access through sensitive registers or memory locations can also be added here. Depending on final APS value of the process, an encrypted packet with process information is broadcasted to all replicas. The overhead of running this algorithm is proportional to: (i) number
of running processes and (ii) size of DOC of each process. The size of the encrypted packages to be shared with replica nodes is directly proportional to the number of control flow instructions in the process they are associated to.

**Algorithm 2** Algorithm for process analysis on node with primary copy of data

<table>
<thead>
<tr>
<th>Require: Compiled Code (DOC) of a process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set-up secure communication among replicas;</td>
</tr>
<tr>
<td>Load APS lookup tables;</td>
</tr>
<tr>
<td>for all process scheduled do</td>
</tr>
<tr>
<td>while current instruction in APS lookup tables do</td>
</tr>
<tr>
<td>get APS of current instruction from APS lookup tables;</td>
</tr>
<tr>
<td>calculate attack probability using Equation 4.1;</td>
</tr>
<tr>
<td>calculate APS of process using Equation 4.2;</td>
</tr>
<tr>
<td>create packet with instruction &amp; data stacks and APS;</td>
</tr>
<tr>
<td>encrypt packet with private key;</td>
</tr>
<tr>
<td>send packet to all replicas;</td>
</tr>
<tr>
<td>go to next process;</td>
</tr>
</tbody>
</table>

Local analysis identifies a possible attack on the primary copy of data. For consensus among replicas to confirm the attack and to take any recovery or prevention steps, the replicas need to be able to securely communicate among themselves. A *complete* attack in a big data system shall modify or delete all copies of a data block residing at primary and replica nodes. Such an attack needs cluster level information about all storage locations of the data block which can only be retrieved from the namenode. A more *probable* attack scenario is to compromise a single data node. In such a case, an attack attempt can be identified by performing simple comparison on data elements such as instruction stack, data stack and APS among primary copy and its replicas. From a security standpoint, a regular communication channel between the infiltrated node and the unaffected replicas is not fully secure. Hence, a communication protocol that uses the dedicated hardware is proposed for secure communication among replica nodes. Similar to other on-board hardware security chips such as TPM chip by TCG[137], the idea is to provide secure keys which
can be used to establish dedicated communication channels among nodes. A unique hardcoded master key that will be used to generate all public-private key pairs for data encryption and decryption. Once the DOC of a process is analyzed, the proposed framework computes an APS associated to the process. For complete security, every process in the system is evaluated.

Next, process analysis information of the process under suspicion is shared securely with other replicas to confirm or deny an attack. A packet for this process is created on the node with the primary copy of data which contains the instruction stack, data stack and APS of the process. The packet is encrypted with a private key and the associated public key is shared with the replicas. This public-private key pair is unique to a node. Periodically updating the public-private key pair of a node is a design decision open to the users.

Figure 4.2: Communication Protocol in the Proposed Framework.
From figure 4.2, it can be noticed that the proposed framework mainly contains 4 parts: static memory for lookup tables, dynamic memory for data and instruction stacks, TPM chip that can generate keys and finally a processor to take care of process monitoring and secure communication. It can be observed from figure 4.2 that the proposed framework works in parallel (dotted lines) with the primary processors of a big data system. The figure shows how the communication protocol fits in a common workflow of a hadoop cluster. First, the namenode receives a job request \texttt{proc1} from the users to which three identical jobs \texttt{proc1.1}, \texttt{proc1.2} and \texttt{proc1.3} are scheduled by the namenode on three data nodes that host the required data and replicas in the hadoop cluster. These regular communication channels are indicated by solid red lines in figure 4.2. When using the proposed framework, these processes undergo process analysis at data nodes and packets of information about the processes are created in case of an attack suspicion. For secure transfer of those packets within the associated replicas in the cluster, the primary data node shares public keys represented by \texttt{pub1} with the replica data nodes. Each data node also shares another set of public keys represented by \texttt{pub2} with the namenode for secure data node to namenode communication. The \texttt{pub2} keys are used in case of \textit{complete} attacks and recovery steps.

This algorithm runs inside the custom coprocessor hardware as soon as a packet is received from replica. The idea behind this algorithm is to perform a simple comparison that runs at multiple levels for optimization. For example, if the size of instruction stack received does not match the size of corresponding local instruction stack, then the algorithm confirms the attack without checking the contents of the stacks. Depending on result from this algorithm, either the process is evaluated as a safe program or necessary recovery actions are to be taken. For this work, steps for attack recovery were not explored because they depend on various system, data and user choices.
Algorithm 3 Algorithm for attack confirmation on replicas

Require: Data packet from replica node

while Packet from replica do
    packet is decrypted with public key;
    instruction stack, data stack and APS are recovered;
    if APS are same then
        if stack sizes are same then
            if instruction stack are same then
                if data stack are same then
                    everything alright;
                else
                    process data is modified;
            else
                process instructions are modified;
        else
            process is modified or not same;
    else
        process is attacked;

4.2.2 Experimental Results

In this section, the setup and design of the experiments and their results are discussed in detail. Most security approaches account for slowdown but with the proposed framework, there is no slowdown in the processing of jobs due to its independent and on-the-fly nature. Hence, the overhead of having additional hardware is measured in terms of parameters such as time, cpu and memory.

Workflow of the proposed framework is dependent on the data hosted by nodes and independent to the job execution cycle. This makes it easy to setup the test environment because only a small subset of the cluster is required to test the end-to-end workflow of the framework. A hadoop cluster of three virtual nodes was established to test the overhead due to the proposed framework. This three-virtual-node hadoop cluster represents a big data system with replication factor of three i.e. primary node and two replicas. Table 4.2 shows the hardware and software configuration of
Table 4.2: Node Configuration.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>AMD A8-6500 APU @ 3.49GHz (1 Core)</td>
</tr>
<tr>
<td>Motherboard</td>
<td>Oracle VirtualBox v1.2</td>
</tr>
<tr>
<td>Chipset</td>
<td>Intel 440FX- 82441FX PMC</td>
</tr>
<tr>
<td>Memory</td>
<td>3072MB</td>
</tr>
<tr>
<td>Disk</td>
<td>215GB VBOX HDD</td>
</tr>
<tr>
<td>Graphics</td>
<td>InnoTek VirtualBox</td>
</tr>
<tr>
<td>Network</td>
<td>Intel 82540EM Gigabit</td>
</tr>
<tr>
<td>OS</td>
<td>CentOS 6.4</td>
</tr>
<tr>
<td>Kernel</td>
<td>2.6.32 – 358.el6.x86_64-x86_4(x86_4)</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.4.7 20120313 ext4</td>
</tr>
<tr>
<td>File-System</td>
<td>ext4</td>
</tr>
<tr>
<td>System Layer</td>
<td>VirtualBox 4.2.16</td>
</tr>
</tbody>
</table>

A node used in the virtual hadoop cluster setup. The proposed framework was tested against 14 benchmark tests that represent memory, processor and system tests. These tests belong to the cpu and crypto suite from Phoronix Test Suite benchmarks[138]. Experiments were designed to first create a baseline using an existing big data system and and then check the overhead of using the proposed framework on the same system.

First, a baseline is created by running the cpu and crypto benchmarks of Phoronix Test Suite on the cluster. The list of 14 tests used to test the proposed framework, along with their metrics, results and characteristics are given in table 4.3. Columns execution, memory and % cpu together talk about the characteristics of baseline. The tests are listed in increasing order of code size. To emulate workflow of a big data system, these tests are run in sequence on the virtual cluster i.e. first on the primary node and then on the replicas.

Assembly code of the 14 benchmark tests was obtained by disassembling their executable files. These assembly files are then given as input to the proposed framework for analysis and attack identification. For convenience, the communication protocol uses basic XOR for packet encryption.
and decryption. TCP was used for communication among the replicas. The framework considers all benchmark tests as potential threats and runs the algorithms on all of them. The lookup tables used to calculate APS used table 4.1 on a predefined list of control flow instructions (given below) and APS values of each of these instructions was set to three. Control flow instructions used are:

- Branch: bt, bts, btr, bswap
- Jump: js, je, jne, jmp, jmpq, ja, jae, jb, jbe, jg, jge, jl, jle, jns, jp, jnp, jo, jno, jrcxz
- Other: call, ret

Results of each test in the baseline are given in table 4.3. These results are also uploaded to the openbenchmark website. The metrics and average result column are generated by the benchmark suite. For this work though, the focus is on the cpu, memory and time statistics of the tests in the benchmark suite as they represent the characteristics of the benchmarks. It can be noticed that the crypto and cpu benchmark tests used in experiments have varying cpu usage (between 35% to 99%). Also, the execution time of these tests varies from low (12 seconds) to high (740 seconds). This variation in time and processing needs of the benchmarks is intentional because an important goals of the proposed framework is to be generic and hence cannot be tested on adhoc experiments.

Columns of table 4.3 starting from column %CFI characterize the overhead of using the proposed framework. The %CFI column shows the significance of control flow instructions in a program. It can be noticed here that control flow instructions typically factor for 10% of the total instructions in a benchmark test. The APS column is derived from %CFI column. Each CFI was given an APS value of 3 and Equations 4.1 and 4.2 are used to calculate the values. The next column i.e. Packet size shows the size of encrypted messages transferred in the network during
the runtime of the proposed framework. This column indicates the network overhead of using the framework on top of a big data system. Since packets contain all information about control flow instructions in a program, their size is proportional to the previous column, %CFI in a benchmark test. One interesting learning from this result is to contemplate the idea of using the framework at function level instead of process level to reduce the packet size.

4.2.3 Analysis of Algorithm and Results

Analysis of algorithm is focused on the limitations of the proposed approach. As mentioned in the previous sections, the rule engine and the APS scoring mechanism used in this technique are premature. Also, this technique can only be applied in homogenous clusters where nodes use the same instruction set architecture. There is no scope for false positives but that is under that assumptions made about attacks and the insiders in the attack model.

Analysis of results is focused on measuring the overhead due to the proposed approach. Time measurements are extremely important when estimating overhead in big data systems. Analysis and confirmation columns of table 4.3 show time consumption of the proposed framework for local process analysis and attack confirmation respectively. It can be noticed from the graph in figure 4.3a that the time required to analyze a process DOC is negligible compared to execution time. The horizontal blue line running along the x-axis of figure 4.3a represents this. The time for network transfer is not included here because it depends on the transfer capacity of the network. The time required for replicas to confirm (or deny) an attack by comparing the packet information with local copy is marginally low when compared to its analysis counterpart. The same can be observed from the graph in figure 4.3b. The horizontal red line running along the x-axis of figure 4.3b represents this. The main reason for such low values in confirmation time is because of multi-level decision
Table 4.3: Phoronix Test Suite Benchmark Results.

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark Test</th>
<th>Execution (seconds)</th>
<th>Memory (mb)</th>
<th>% CPU</th>
<th>% CFI</th>
<th>APS (mb)</th>
<th>Packet size (seconds)</th>
<th>Analysis (seconds)</th>
<th>Consensus (seconds)</th>
<th>% Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stream</td>
<td>12</td>
<td>1023</td>
<td>40%</td>
<td>5%</td>
<td>-0.89</td>
<td>0.004</td>
<td>0.01</td>
<td>0.01</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>C-Ray</td>
<td>322</td>
<td>1011</td>
<td>98%</td>
<td>9%</td>
<td>-0.64</td>
<td>0.012</td>
<td>0.02</td>
<td>0.02</td>
<td>0.0001</td>
</tr>
<tr>
<td>3</td>
<td>Smallpt</td>
<td>740</td>
<td>665</td>
<td>99%</td>
<td>5%</td>
<td>-0.89</td>
<td>0.017</td>
<td>0.01</td>
<td>0.03</td>
<td>0.00005</td>
</tr>
<tr>
<td>4</td>
<td>Himeno Benchmark</td>
<td>65</td>
<td>1029</td>
<td>89%</td>
<td>8%</td>
<td>-0.68</td>
<td>0.026</td>
<td>0.02</td>
<td>0.02</td>
<td>0.0006</td>
</tr>
<tr>
<td>5</td>
<td>Apache Benchmark</td>
<td>172</td>
<td>1116</td>
<td>95%</td>
<td>13%</td>
<td>-0.49</td>
<td>0.054</td>
<td>0.02</td>
<td>0.03</td>
<td>0.0002</td>
</tr>
<tr>
<td>6</td>
<td>TSCP</td>
<td>20</td>
<td>1026</td>
<td>35%</td>
<td>14%</td>
<td>-0.48</td>
<td>0.079</td>
<td>0.02</td>
<td>0.02</td>
<td>0.002</td>
</tr>
<tr>
<td>7</td>
<td>Parallel BZIP2 Compression</td>
<td>59</td>
<td>946</td>
<td>85%</td>
<td>13%</td>
<td>-0.5</td>
<td>0.20</td>
<td>0.16</td>
<td>0.04</td>
<td>0.003</td>
</tr>
<tr>
<td>8</td>
<td>FLAC Audio Encoding</td>
<td>14</td>
<td>729</td>
<td>75%</td>
<td>14%</td>
<td>-0.48</td>
<td>0.42</td>
<td>0.22</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>9</td>
<td>John The Ripper</td>
<td>70</td>
<td>1026</td>
<td>92%</td>
<td>9%</td>
<td>-0.63</td>
<td>0.6</td>
<td>0.37</td>
<td>0.04</td>
<td>0.006</td>
</tr>
<tr>
<td>10</td>
<td>LAME MP3 Encoding</td>
<td>23</td>
<td>729</td>
<td>82%</td>
<td>10%</td>
<td>-0.61</td>
<td>0.89</td>
<td>0.46</td>
<td>0.05</td>
<td>0.02</td>
</tr>
<tr>
<td>11</td>
<td>Crafty</td>
<td>124</td>
<td>670</td>
<td>94%</td>
<td>7%</td>
<td>-0.76</td>
<td>1.35</td>
<td>1.06</td>
<td>0.07</td>
<td>0.009</td>
</tr>
<tr>
<td>12</td>
<td>7-Zip Compression</td>
<td>40</td>
<td>1015</td>
<td>82%</td>
<td>13%</td>
<td>-0.5</td>
<td>2.95</td>
<td>1.4</td>
<td>0.12</td>
<td>0.03</td>
</tr>
<tr>
<td>13</td>
<td>Graphics Magick</td>
<td>62</td>
<td>1028</td>
<td>89%</td>
<td>12%</td>
<td>-0.53</td>
<td>4.88</td>
<td>2.54</td>
<td>0.18</td>
<td>0.04</td>
</tr>
<tr>
<td>14</td>
<td>OpenSSL</td>
<td>22</td>
<td>730</td>
<td>72%</td>
<td>11%</td>
<td>-0.55</td>
<td>6.82</td>
<td>2.77</td>
<td>0.23</td>
<td>0.13</td>
</tr>
<tr>
<td><strong>Average Values</strong></td>
<td><strong>121</strong></td>
<td><strong>910</strong></td>
<td><strong>81%</strong></td>
<td><strong>10%</strong></td>
<td><strong>-0.6</strong></td>
<td><strong>1.30</strong></td>
<td><strong>0.65</strong></td>
<td><strong>0.06</strong></td>
<td><strong>0.017</strong></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.3: Time Overhead for APS Technique.

(a) Process Analysis vs Execution Time

(b) Local Analysis (step 1) vs Comparison (step 2)
making and using separate instruction and data stacks. But the most interesting observation from these tests is that the time taken by the framework to determine whether a process is attacked or not. This is calculated by adding the values for analysis and confirmation; and dividing the result with the execution time. The average time taken by the proposed framework to detect an attack is 0.01% of the time required to complete the corresponding benchmark test. Also, on average, the proposed framework uses 85% of the cpu for process analysis and 94% of the cpu for attack confirmation.

4.3 Technique Based on Control Flow Graphs Analysis

Though the central theme to all the proposed compile-time techniques is to reduce the amount of work needed to analyze a program binary, the two techniques proposed until now are heuristic in nature. In this approach of detecting attacks based on Control Flow Graph (CFG) analysis, the problem of graph isomorphism to check CFG similarity is handled using a heuristic approach.

4.3.1 Proposed Methodology

The first step in this technique involves capturing the control-flow of a process running on a datanode of the big data system. This is realized by converting a CFG to a set of Minimum Spanning Trees (MST). The second step involves process-level similarity check followed by consensus among replica datanodes. The similarity check in this case involves an exhaustive matching among sets of MSTs.

In this work, the emphasis is on process level intrusion detection by observing coherence in the behavior of duplicate processes running on replica datanodes of a distributed big data system.
To capture the program behavior, the first step is to identify a representation of the program that has the required information and filters out all other data. Such representation is called the program signature. Since the goal is to identify intrusions from control-flow mismatch, program signatures should contain all possible control flow information of a program.

Compiled source code of a program is generally used to generate static CFG. Since most big data frameworks use a virtual machine (such as JVM), an instruction level CFG in this context is generated from java byte code. In this work, disassembled object code (DOC) from java byte code is used as input to generate the CFG at instruction level. It is important for the program signature to contain only the information that is necessary. Hence, every CFG is converted into a set of MSTs that are later used to generate the program signature. In this work, the idea of representing a program by a set of MSTs/MSAs is proposed. Such a set of MSTs can be extracted from a byte-level CFG using Edmonds algorithm. This set of MSTs that are extracted from a CFG are further filtered to only the set of edge-disjoint MSTs. There are many versions proposed for Edmonds algorithm [96, 97, 98] and for this work a version from NetworkX graph library [139] is used that generates edge-disjoint spanning trees from the root vertex of a given digraph. Once a minimal representation of the logic in a program is obtained in the form of an MSA, it is converted into a string which is in accordance to the DOT format representation.
The length of a MST string in DOT format is dependent on program size because DOT format is given by listing all the nodes of MSA followed by the edges. To make the comparison step faster, the variable length MST strings of a program are converted to fixed length strings using hashing. The extracted set of edge-disjoint MSTs are hashed using popular hashing algorithms such as SHA or MD5 to generate a set of fixed-length hash strings. Since a sparse graph such as CFG can have multiple MSAs, the program signature can be a single hash string or a set of hash strings. Having all possible MSAs in the program signature makes the graph similarity check more reliable. In the end, a program signature is a set of fixed-length strings.

Program signatures are encrypted before being shared with replica datanodes for tighter security. The private key for encryption is generated from a hardcoded master key of secure hardware, similar to the idea proposed previously [19]. Every datanode in a big data system runs the proposed profiling method for every running process and it includes all the steps involved in converting the compiled binary of a program to its program signature. A pictorial representation of the steps in profiling method is given in Figure 4.4.

Replication property of big data systems opens scope for new methods of implementing application logic level IDS techniques. Process similarity check among duplicate nodes of the cluster helps in checking for coherence among the replica datanodes while performing a write or read operation. When a process is scheduled to run on a datanode that hosts the primary copy of a data, a signature for that process is created by the profiling method (Step 1) of the proposed IDS technique and that signature string is shared with all replica datanodes. In the matching method (Step 2), these signatures received from other datanodes are decrypted and matched with the local versions of the same process. The results are shared with all other replica datanodes for consensus. For
secure communication among datanodes, the same communication protocol is used that is proposed in the security framework.

A crucial part of the matching method is to check for similarity (or dissimilarity) between two program signatures. Generally, graph similarity check can be performed by checking node similarity and edge similarity. The following points are considered while comparing MSTs to check for similarity among programs:

- MSTs are sparse graphs obtained from byte-level CFGs. Hence, checking for path sensitivity is not exponential.

- All edges are assumed to have the same weight of 1.

- The total number of MSTs for a CFG is limited (by Cayley’s formula [140]).

- According to Edmond’s theorem, a graph which is k-connected always has k edge-disjoint arborescences.

- Two MSTs are a perfect match if their node sets and edge sets match exactly.

- If edge set of one MST is a subset of the edge set of another MST, the source graphs of these MSTs are not similar.

- Two graphs are similar if for every MST in one graph there exists a perfect match in the set of MSTs of the other graph.

- Hashing algorithms such as SHA1 or MD5 are quick and efficient.

Based on the points listed above, the following method is developed for graph similarity check. Let us consider 2 control-flow graphs G1 and G2. Let \((N1, E1)\) represent G1 where N1 is
the node set of the graph \( G_1 \) and \( E_1 \) is the edge set of the graph. Similarly, \( (N_2, E_2) \) represents \( G_2 \) where \( N_2 \) is the node set of the graph \( G_1 \) and \( E_2 \) is the edge set of the graph. Assume that \( M_1 [(N_1, E_1')] \) is the set of MST/MSA for \( G_1 \) and \( M_2 [(N_2, E_2')] \) is the set of MST/MSA for \( G_2 \) obtained after applying a variation of Edmonds algorithm on these CFGs (such as finding all edge-disjoint MSTs). In order to check for similarity in both graphs \( G_1 \) and \( G_2 \), a perfect match in \( M_2 \) for all MSTs in \( M_1 \) is required. To simplify the match function, a hash function is used on \( M_1 \) and \( M_2 \) that creates a unique hash for every MST. Let \( H_1 \) be a set of hashes generated from \( M_1 \) and \( H_2 \) be the set of hashes from \( M_2 \). If any hash in \( H_1 \) does not exist in \( H_2 \), it can be deduced that the graphs are not equal.

### 4.3.2 Experimental Results

In this section, the experimental setup and experiments used for testing the proposed technique are provided. The results and some analysis are also provided.

An Amazon EC2 [141] m4.xlarge instance running Ubuntu 14.04 is used to generate MSTs (and their hashes) from CFGs using SageMath. The proposed technique was implemented and tested on an Amazon EC2 big data cluster of 5 t2.micro nodes - 1 master node, 1 secondary master node and 3 datanodes with a replication factor of 3. The list of softwares used in conducting the experiments are:

- SageMath [142] is a free open-source mathematics software system for mathematical calculations.

- GraphML [143] is a popular graph representation format which can used to represent both CFG and MST.
• Graphviz [144] is open source graph visualization software that takes input in DOT format and makes diagrams in useful formats.

• NetworkX [139] is a Python language software package that provides graph algorithms such as Edmonds and VF2.

• Control-flow graph factory [145] is a software that generates CFGs from java bytecode (class file) and exports them to GraphML or DOT formats.

The proposed intrusion detection technique was tested using 16 hadoop MapReduce examples that can be found in all hadoop distributions. These examples cover a wide range of big data applications as listed in Table 4.6. The class files of these examples are readily available in the hadoop distributions. First, control-flow graph factory [145] was used to generate control flow graphs from the class files. These graphs are stored in graphml format and given as input to a simple SageMath [142] script that uses NetworkX library [139] and computes the edge-disjoint MSAs and hashes them using MD5. A C++ application was used to implement encryption and secure communication needed for the proposed IDS technique. The implementation was based on framework from [19]. Since the hashes are fixed length strings, a basic numeric key based left/right shift is sufficient for encryption/decryption of messages. Some of the MapReduce examples that do not have set benchmarks were executed with minimum input requirements.

Table 4.4, Figures 4.5a and 4.5b show the results of the experiments. Figure 4.5a shows the comparison between the time taken to run the Hadoop MapReduce examples on a big data cluster and the time taken to run the proposed intrusion detection technique. The execution times for some examples (represented by * in table 4.4) are inconsistent among multiple runs. It can be noticed from table 4.4 that only 0.81% of time taken to execute an example is needed to analyze
(a) Proposed IDS Technique and Run-time for MapReduce Examples.

(b) Profiling and Matching Methods of the Proposed IDS Technique.

Figure 4.5: Execution Time Analysis of CFG Technique.
Table 4.4: Program-Level Time Metrics of Hadoop MapReduce Examples (in Seconds)

<table>
<thead>
<tr>
<th>E.No</th>
<th>Example</th>
<th>Profile Method</th>
<th>CFG to MSA set</th>
<th>Hash Method</th>
<th>Match Method</th>
<th>Avg. Hash Match</th>
<th>Consensus</th>
<th>Total Time</th>
<th>Exec. Time</th>
<th>% Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>word mean</td>
<td>0.0216</td>
<td>0.0216</td>
<td>7.89E-05</td>
<td>0.0190</td>
<td>0.0002</td>
<td>0.0187</td>
<td>0.0407</td>
<td>6.988</td>
<td>0.58%</td>
</tr>
<tr>
<td>2</td>
<td>pentomino</td>
<td>0.0288</td>
<td>0.0288</td>
<td>8.70E-05</td>
<td>0.0196</td>
<td>0.0013</td>
<td>0.0182</td>
<td>0.0485</td>
<td>4.914</td>
<td>0.99%</td>
</tr>
<tr>
<td>3</td>
<td>distributed bbp*</td>
<td>0.0567</td>
<td>0.0567</td>
<td>6.29E-05</td>
<td>0.0150</td>
<td>0.0019</td>
<td>0.0130</td>
<td>0.0718</td>
<td>28.58</td>
<td>0.25%</td>
</tr>
<tr>
<td>4</td>
<td>aggregate wordcount</td>
<td>0.0070</td>
<td>0.007</td>
<td>5.70E-05</td>
<td>0.0145</td>
<td>0.0002</td>
<td>0.0143</td>
<td>0.0215</td>
<td>19.002</td>
<td>0.11%</td>
</tr>
<tr>
<td>5</td>
<td>secondary sort*</td>
<td>0.0199</td>
<td>0.0199</td>
<td>5.10E-05</td>
<td>0.0072</td>
<td>0.0018</td>
<td>0.0054</td>
<td>0.0272</td>
<td>11.657</td>
<td>0.23%</td>
</tr>
<tr>
<td>6</td>
<td>aggregate word histogram</td>
<td>0.0066</td>
<td>0.0066</td>
<td>4.20E-05</td>
<td>0.0135</td>
<td>0.0012</td>
<td>0.0122</td>
<td>0.0201</td>
<td>18.024</td>
<td>0.11%</td>
</tr>
<tr>
<td>7</td>
<td>random writer</td>
<td>0.2561</td>
<td>0.2561</td>
<td>8.58E-05</td>
<td>0.0217</td>
<td>0.0025</td>
<td>0.0191</td>
<td>0.2779</td>
<td>29.111</td>
<td>0.95%</td>
</tr>
<tr>
<td>8</td>
<td>tera validate</td>
<td>0.0181</td>
<td>0.0181</td>
<td>5.20E-05</td>
<td>0.0169</td>
<td>0.0001</td>
<td>0.0168</td>
<td>0.0351</td>
<td>5.958</td>
<td>0.59%</td>
</tr>
<tr>
<td>9</td>
<td>qmc*</td>
<td>0.0238</td>
<td>0.0238</td>
<td>7.39E-05</td>
<td>0.0202</td>
<td>0.0015</td>
<td>0.0186</td>
<td>0.0440</td>
<td>11.657</td>
<td>0.38%</td>
</tr>
<tr>
<td>10</td>
<td>word standard deviation</td>
<td>0.0193</td>
<td>0.0193</td>
<td>7.89E-05</td>
<td>0.0098</td>
<td>0.0021</td>
<td>0.0076</td>
<td>0.0292</td>
<td>7.112</td>
<td>0.41%</td>
</tr>
<tr>
<td>11</td>
<td>word median</td>
<td>0.0312</td>
<td>0.0312</td>
<td>6.20E-05</td>
<td>0.0208</td>
<td>0.0020</td>
<td>0.0187</td>
<td>0.0520</td>
<td>7.028</td>
<td>0.73%</td>
</tr>
<tr>
<td>12</td>
<td>bbp</td>
<td>0.0415</td>
<td>0.0415</td>
<td>9.08E-05</td>
<td>0.0118</td>
<td>0.0003</td>
<td>0.0115</td>
<td>0.0534</td>
<td>6.865</td>
<td>0.78%</td>
</tr>
<tr>
<td>13</td>
<td>tera gen</td>
<td>0.0169</td>
<td>0.0169</td>
<td>5.51E-05</td>
<td>0.0131</td>
<td>0.0023</td>
<td>0.0108</td>
<td>0.0301</td>
<td>4.905</td>
<td>0.61%</td>
</tr>
<tr>
<td>14</td>
<td>sudoku*</td>
<td>0.0177</td>
<td>0.0177</td>
<td>5.60E-05</td>
<td>0.0156</td>
<td>0.0006</td>
<td>0.0150</td>
<td>0.0334</td>
<td>11.657</td>
<td>0.29%</td>
</tr>
<tr>
<td>15</td>
<td>wordcount</td>
<td>0.3672</td>
<td>0.3672</td>
<td>6.99E-05</td>
<td>0.0221</td>
<td>0.0023</td>
<td>0.0197</td>
<td>0.3893</td>
<td>7.034</td>
<td>5.54%</td>
</tr>
<tr>
<td>16</td>
<td>multfile wordcount</td>
<td>0.0159</td>
<td>0.0159</td>
<td>5.20E-05</td>
<td>0.0118</td>
<td>0.0001</td>
<td>0.0116</td>
<td>0.0277</td>
<td>5.963</td>
<td>0.47%</td>
</tr>
<tr>
<td><strong>Average Values</strong></td>
<td><strong>0.0593</strong></td>
<td><strong>0.0592</strong></td>
<td><strong>6.59E-05</strong></td>
<td><strong>0.0158</strong></td>
<td><strong>0.0013</strong></td>
<td><strong>0.0144</strong></td>
<td><strong>0.07516</strong></td>
<td><strong>11.657</strong></td>
<td><strong>0.81%</strong></td>
<td></td>
</tr>
</tbody>
</table>

The time needed to run the proposed detection technique includes (a) time taken to create CFG for the main method from the class file; (b) time taken to extract MST set from CFG; (c) time taken to hash the MSTs and encrypt them and; (d) time taken to check for similarity among duplicate processes by comparing the program signatures.
4.3.3 Analysis of Algorithm and Results

It can be noticed from Figure 4.5b that the time required by the proposed technique is influenced by the profiling method trying to extract MSAs from CFG, particularly when there are more than one MSAs for a CFG. Though the matching method performance is directly proportional to the square of the size of the number of edge-disjoint MSAs in a CFG i.e. $O(n^2)$ worst case complexity, it can be observed that it is rare to have more than a couple of edge-disjoint MSAs in a CFG because of the sparse nature of CFG.

4.4 Technique Based on Control Instruction Sequence Analysis

As mentioned before, the central theme to all compile-time techniques proposed in this work is to reduce the amount of work needed to analyze a program binary. In this approach of detecting attacks based on Control Instruction Sequence (CIS) analysis, a program binary is uniquely represented by preserving the order of control-flow instructions in the disassembled object code. To simplify the verification step, each process is represented by a hash that is obtained from the CIS of that process.

4.4.1 Proposed Methodology

The attack detection algorithm based on CIS is a two step process: process profiling (step 1) and consensus through hash matching (step 2).

Traditionally vulnerability scanning is performed away from the source program’s execution domain to guarantee isolation. Hence, the results of such scans must be communicated back to the program. But this leads to a cost versus isolation trade-off, depending on the remoteness of
Algorithm 4 Process Profiling

procedure Process Profile
  while true do
    procnew ← get newly scheduled process
    code ← get assembly code from HotSpotVM(procnew)
    for instr ∈ code do
      if instr ∈ jump then
        seqjump ← add instr to sequence of jumps
      if instr ∈ call then
        seqcall ← add instr to sequence of calls
      if instr ∈ return then
        seqreturn ← add instr to sequence of returns
    seqarray ← add seqjump
    seqarray ← add seqcall
    seqarray ← add seqreturn
    for seq ∈ seqarray do
      hashseq ← get hash from sha(seq)
      hashhashes ← add hashseq
      msg ← get hash from sha(hashhashes)
      send msg using Secure Communication Protocol
  end

the location used to perform the vulnerability scan. In big data applications, the source program’s
execution is distributed across multiple nodes of the cluster. This makes it difficult to use techniques
such as vulnerability scans because of the additional communication and synchronization involved.
Hence it is imperative to look at properties of big data frameworks that can be exploited to provide
security. For example, big data infrastructures use replication of data for high availability. Cons-
istency requirements enforce the same program to be run on multiple nodes that host the data
required for the program. This unique property of big data systems is used to propose a novel
process profiling technique which is a variation of CFI and can help in detecting insider attacks
within the big data system. Evans et al. [110] show that CFI, either with limited number of tags
or unlimited number of tags, is not completely effective in attack prevention because it is context
insensitive. Also, CFI is usually based on CFG created from static analysis of program code.
During the initial days of big data, applications were packaged as jars that run on Java Virtual Machines (JVM). These jars are not completely compiled and do not convey much about the program they represent. Hence, in this work CFI is not used for CFG’s created using statistical code analysis. Instead, the control structure of a program is built from the corresponding JVM output i.e. the assembly code of the Hotspot VM that hosts the JVM. Since this is considered the final run-time code that gets executed on the hardware, the control structure generated from the output of Hotspot VM is expected to be less susceptible to software attacks compared to a CFG generated from statistical analysis of program code. In the context of big data platforms, this mitigates the possibility of launching an attack on the entire cluster. Another major variation from CFI in this process profiling technique is to use individual control flow instruction sequences instead of CFG paths. Control instructions dictate the control flow in a program. Generating instruction sequences of such control flow instructions from the assembly code output of Hotspot VM should technically give us all information a CFG can provide in this context and avoid the complexity involved in generating a CFG. Technically, the graph matching problem is converted to a much simpler string matching problem. Of course, newer big data applications are moving away from using JVMs to other programming domains such as python and we have not covered them in this work.

The analyzer module in the proposed system creates instruction sequences for jumps, calls and returns from the JVM output of a given program (based on Intel’s Instruction Set Architecture). Then, the SHA cryptographic hash function module is used to generate a fixed-length output for each of the three instruction sequences. All three hashes are combined and again given to the SHA cryptographic hash function module to generate a final hash for the program. This hash of hashes
Algorithm 5 Hash Matching

procedure Hash Match
    while true do
        msg\_p ← get message about process p from main copy
        hash\_hashes\_received\_p ← decrypt(msg\_new, priv\_k)
        hash\_hashes\_local\_p ← process\_profile(p)
        if hash\_hashes\_received\_p ∈ hash\_hashes\_local\_p then
            confirmation ← safe
        else
            confirmation ← unsafe
        send(confirmation, main)

strengthened the uniqueness in identifying a program. All programs that run on every node in the
cluster will follow the same routine. Encryption module of the node with the primary copy of data
uses currently active public keys of replica nodes to encrypt the hash of hashes and send it to the
associated replica node. Hence, this node acts as the coordinator for performing step 2 in the attack
detection algorithm.

Algorithm 4 shows the steps involved in the proposed process profiling step. This algorithm
will be running independently in the analyzer module of all machines in the big data cluster. Every
process output, proc\_new, from the HotSpot VM is grabbed by the analyzer module of the proposed
system and profiled based on the control flow instructions present in its assembly code. Line by
line analysis of proc\_new is conducted and each instruction instr is matched with the set of control
flow instructions available in the instruction set of the processor architecture. For this work, only
the most prominent control flow instructions of Intel’s x86 architecture are used: jumps, calls and
returns. When an instr in the code of the proc\_new is a control flow instruction, it gets added
to the corresponding sequence string. The seq\_array represents the array of individual control flow
instruction sequences in the process proc\_new. This array is used later as input while generating the
hashes for each control sequence string. All fixed length hash outputs are combined as hash\_hashes and
rehashed to generate a final hash called $msg$ that represents the program. This $msg$ is then shared with all replicas running the same program using the secure communication protocol described above.

The next step in the attack detection algorithm is a consensus algorithm similar to the extended 2-phase commit protocol [146]. In this step, the node with primary copy of data acts as coordinator and requests all replica nodes, that act as workers, to confirm if their local hash of hashes, $(msg)$ of a particular process matches exactly with the coordinator’s version. The coordinator then decides on the safety of the process depending on the acknowledgments received from participating replica nodes. A process is considered to be safe by the coordinator if and only if it receives safe acknowledgments from all of the workers. At the end of process profiling step, encrypted message $msg_e$ is shared by coordinator node with all worker nodes. The nodes that receive such messages will
decrypt the message with their currently active private key. The decrypted message is essentially
the hash of hashes of the three control instruction sequence strings. This decrypted hash of hashes
can be directly compared to the local version of the same process to detect the possibility of an
attack.

If the result of such comparison of strings is a perfect match, then that indicates that the
same process (with the same code) was run on both nodes. This indicates a safe process unless
both nodes of the cluster are attacked the same way, in which case it will be a false positive. A
confirmation message about the result of the hash comparison will be sent to the coordinator node
as response to the original incoming message. The coordinator node will wait to receive responses
from all replicas in order to arrive at a conclusion about the possibility of an attack in a process.
The given big data system is safe as long as all the replicas respond with a safe confirmation. A
single unsafe response will mean that the system is under attack. Algorithms 5 and 6 give more
details about the hash matching and consensus steps that take place in step 2 of the attack detection
algorithm. A pictorial representation of the steps involved in the 2-step attack detection algorithm
is given in Figure 4.6. This figure represents a big data system with a replication factor of 3 and
hence there is one coordinator (represented with a dark black shadow below the node) and two
workers. Active communication channels are represented using a dotted line while the regular lines between nodes represent passive communication channel. The blue dotted loop around each node in step 1 and 3 of the figure represent local computations.

Algorithm 5 is used in the hash matching step of the attack detection algorithm. When a worker node, $node_k$, receives $msg_p$ from the coordinator node about a process $p$, it will decrypt that message using its current private key, $priv_k$ and stores the result as $hash_{hashes}(received_p)$. The local version of the same string i.e. $hash_{hashes}(local_p)$ will be compared against the $hash_{hashes}(received_p)$ to identify similarity between local and received hash of a process. The result of this hash matching is sent back as confirmation to the coordinator node, main. The value of confirmation is safe in case of a perfect match of hashes and unsafe otherwise.

Algorithm 6 is used by the coordinator node to identify an attack, with the help of worker nodes. After step 1, the coordinator node waits for responses from all the recipients. The worker nodes respond with a confirmation message that says whether the process is safe or unsafe. If the count of number of safe responses i.e. $count_{safe}$ from worker nodes matches with the count of number of nodes in the replica set i.e. $count_{replicas}$, the coordinator node assumes that there is no attack in the current process $p$ and resets the attack variable. Else, if a mismatch in the process analysis is observed, the attack variable is set and the master node is notified about the possibility of an attack in process $p$.

### 4.4.2 Experimental Results

In this section the experimental setup is described; details about the choice of experiments are explained; and the results and their analysis are presented. The hadoop security design specifies
that a 3% slowdown in performance is permissible for any newly proposed security solutions [75]. Hence, it is important for the proposed system to offer both theoretical correctness and feasibility in practical implementation and usage. Security in big data systems is a new area that does not have set standards and specifically designed open-source benchmarks to evaluate the overhead. Hence, a set of general big data benchmark programs that are relevant and provided by the big data community were handpicked to test the efficiency of the proposed security system. The 3 big data services used for experiments are:

- Hadoop [8], the most popular implementation of a big data framework that is maintained by the Apache open-source community. It allows storing and processing of large data using programming models such as MapReduce.

- Spark [9], a fast and general engine for large-scale data processing that is supposedly much faster than Hadoop and it is maintained by the Apache open-source community as well.

- Amazon web services (AWS) [141, 147, 148], a perfect example of real-world big data system. AWS provides Elastic Cloud Compute (EC2) service that allows users to use Amazon cloud’s compute capacity depending on their needs. EC2 presents a true virtual computing environment. Storage for the EC2 nodes is provided by Amazon Elastic Block Store (EBS) which offers persistent storage. EBS volumes are automatically replicated to protect user from component failure, offering high availability and durability.

AWS supported hadoop and spark clusters were used for conducting the experiments. The
\textit{Hadoop Cluster} is a 5 node cluster built using basic t2.micro nodes of Amazon EC2 and EBS. Each node is equipped with only 1 vCPU and 1GB memory. The network performance is minimal for this
Table 4.5: Amazon EC2 Instance Types

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance Model</td>
<td>t2.micro</td>
</tr>
<tr>
<td></td>
<td>m1.large</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Xeon with Turbo</td>
</tr>
<tr>
<td></td>
<td>Intel Xeon E5-2650</td>
</tr>
<tr>
<td>Compute Units</td>
<td>1 (Burstable)</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>vCPU</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Memory (GB)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7.5</td>
</tr>
<tr>
<td>Storage (SSD)</td>
<td>Elastic Block Store</td>
</tr>
<tr>
<td></td>
<td>Elastic Block Store</td>
</tr>
<tr>
<td>Networking Performance</td>
<td>low</td>
</tr>
<tr>
<td></td>
<td>moderate</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux/UNIX</td>
</tr>
<tr>
<td></td>
<td>Linux/UNIX</td>
</tr>
<tr>
<td>Hadoop distribution</td>
<td>2.7.1</td>
</tr>
<tr>
<td></td>
<td>2.7.1</td>
</tr>
<tr>
<td>Spark distribution</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>1.6</td>
</tr>
</tbody>
</table>

The Spark Cluster is a 4 node cluster built using general purpose m1.large nodes of Amazon EC2 and EBS. Each node is equipped with 2 vCPU and 7.5GB memory. Network performance is moderate for this cluster. Both cluster configurations satisfy the minimum requirement to support replication factor of 3. The hardware and software configurations of the EC2 nodes can be found in table 4.5. A 64-bit Ubuntu AMI (Amazon Machine Instance) is built for each node-type before setting up the clusters. These AMIs were equipped with the latest distributions of Hadoop, Spark and GCC along with with our code base. The hadoop cluster had 5 nodes, where 1 node acted as the namenode, 1 node acted as the secondary name node and 3 nodes were acting as data nodes. The spark cluster had a master and 3 slave nodes. Since the proposed system works independently, all modules of the model had to be installed on every node of the EC2 clusters. A library of all modules in the model was implemented in C++ programming language using STL and multi-threading libraries and packaged together. Our code used TCP/IP protocol and SSH keys for communication between the nodes of the clusters.

Though the main requirement for any attack detection service is to be able to detect an attack successfully, it is also necessary to be able to detect the attack before the attacked program
Table 4.6: List of Hadoop MapReduce Examples

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aggregate word count</td>
<td>An Aggregate-based map/reduce program that counts the words in the input files.</td>
</tr>
<tr>
<td>2</td>
<td>aggregate wordhist</td>
<td>An Aggregate-based map/reduce program that computes the histogram of the words in the input files.</td>
</tr>
<tr>
<td>3</td>
<td>bbp</td>
<td>A map/reduce program that uses Bailey-Borwein-Plouffe to compute exact digits of Pi.</td>
</tr>
<tr>
<td>4</td>
<td>distbbp</td>
<td>A map/reduce program that uses a BBP-type formula to compute exact bits of Pi.</td>
</tr>
<tr>
<td>5</td>
<td>grep</td>
<td>A map/reduce program that counts the matches of a regex in the input.</td>
</tr>
<tr>
<td>6</td>
<td>pi</td>
<td>A map/reduce program that estimates Pi using a quasi-Monte Carlo method.</td>
</tr>
<tr>
<td>7</td>
<td>random text writer</td>
<td>A map/reduce program that writes 10GB of random textual data per node.</td>
</tr>
<tr>
<td>8</td>
<td>random writer</td>
<td>A map/reduce program that writes 10GB of random data per node.</td>
</tr>
<tr>
<td>9</td>
<td>sort</td>
<td>A map/reduce program that sorts the data written by the random writer.</td>
</tr>
<tr>
<td>10</td>
<td>teragen</td>
<td>Generate data for the terasort.</td>
</tr>
<tr>
<td>11</td>
<td>terasort</td>
<td>Run the terasort.</td>
</tr>
<tr>
<td>12</td>
<td>teravalidate</td>
<td>Check the results of the terasort.</td>
</tr>
<tr>
<td>13</td>
<td>word count</td>
<td>A map/reduce program that counts the words in the input files.</td>
</tr>
<tr>
<td>14</td>
<td>word mean</td>
<td>A map/reduce program that counts the average length of the words in the input files.</td>
</tr>
<tr>
<td>15</td>
<td>word median</td>
<td>A map/reduce program that counts the median length of the words in the input files.</td>
</tr>
<tr>
<td>16</td>
<td>word standard deviation</td>
<td>A map/reduce program that counts the standard deviation of the length of the words in the input files.</td>
</tr>
</tbody>
</table>

completes execution. Hence, the efficiency as well as the overhead of the proposed system are shown by conducting the experiments in real-time using a selected set of popular examples and tests from the big data community. Two sets of open source big data benchmark programs are used in this work: (a) 16 Hadoop MapReduce Examples that are provided in the Apache hadoop installation kit; and (b) 16 Spark-perf MLlib Tests for machine learning algorithms given in the spark performance test suite by Databricks [149]. More details about these examples and tests are given in tables 4.6 and 4.7.
Table 4.7: List of Spark-Perf MLlib Tests

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fp-growth</td>
<td>Frequent Pattern Matching Tests to find frequent item sets</td>
</tr>
<tr>
<td>2</td>
<td>word2vec</td>
<td>Feature Transformation Tests for distributed presentation of words</td>
</tr>
<tr>
<td>3</td>
<td>chi-sq-feature</td>
<td>Statistic Toolkit Tests using Chi-square for correlation</td>
</tr>
<tr>
<td>4</td>
<td>spearman</td>
<td>Statistic Toolkit Tests using Spearman’s Correlation</td>
</tr>
<tr>
<td>5</td>
<td>pearson</td>
<td>Statistic Toolkit Tests using Pearson’s Correlation</td>
</tr>
<tr>
<td>6</td>
<td>block-matrix-mult</td>
<td>Matrix Multiplication on distributed matrix</td>
</tr>
<tr>
<td>7</td>
<td>summary-statistics</td>
<td>Linear Algebra Tests using Summary Statistics (min, max, ...)</td>
</tr>
<tr>
<td>8</td>
<td>pca</td>
<td>Linear Algebra Tests using Principal Component Analysis</td>
</tr>
<tr>
<td>9</td>
<td>svd</td>
<td>Linear Algebra Tests using Singular Value Decomposition</td>
</tr>
<tr>
<td>10</td>
<td>gmm</td>
<td>Clustering Tests using Gaussian Mixture Model</td>
</tr>
<tr>
<td>11</td>
<td>kmeans</td>
<td>Clustering Tests using K-Means clustering</td>
</tr>
<tr>
<td>12</td>
<td>als</td>
<td>Recommendation Tests using Alternating Least Squares</td>
</tr>
<tr>
<td>13</td>
<td>decision-tree</td>
<td>Random Forest Decision Tree</td>
</tr>
<tr>
<td>14</td>
<td>naive-bayes</td>
<td>Classification Tests using Naive Bayes</td>
</tr>
<tr>
<td>15</td>
<td>glm-classification</td>
<td>Generalized Linear Classification Model</td>
</tr>
<tr>
<td>16</td>
<td>glm-regression</td>
<td>Generalized Linear Regression Model</td>
</tr>
</tbody>
</table>

The input to the model (built from the proposed system) is the run-time assembly code of a program. The Hadoop MapReduce examples were coded in Java and the Spark-perf MLlib tests were coded in Scala. So, the jars to run these examples were built using just-in-time compiling. Their bytecodes are insufficient to create the assembly codes of the individual programs. A software called jit-watch [150] is used to generate the assembly codes (Intel x86 specification) of the programs from the jars. Since the proposed technique only needs control-flow instructions from the generated assembly code outputs of each program, a custom parser is used that can filter out control flow instructions from the native files. All 32 example programs are infected by a code snippet that calls a function `foo` to print a line to the console and involves a total of 3 call instructions and 1 return instruction. The command used for generating assembly code output of JVM (or Hotspot VM) when running the program is:
First, the execution times for the Hadoop MapReduce examples on the hadoop cluster were measured. Then the run times of the implemented model, while it was analyzing the assembly codes of the driver programs of the same examples, were studied. These experiments are adhoc because the input arguments for some of the experiments were intentionally low to simulate worst case scenarios where the process takes very less time to execute. Data is required for some of the MapReduce examples to execute. To meet such input data requirements of the MapReduce examples, the configuration file data from etc folder of the Hadoop installation is put into the Hadoop database (HDFS). The generic command used to run these MapReduce examples and measure the time taken is given as:

time hadoop jar hadoop-mapreduce-examples.jar [main method] [args]

The spark-perf MLlib tests on the spark cluster were conducted the same way the MapReduce examples were tested. But here the inputs for the tests were predetermined by the benchmark provider in the config.py script. The generic command used to run these MLlib tests is:


4.4.3 Analysis of Algorithm and Results

The experiments used for evaluating the proposed security system comprise of stress tests and performance benchmarks of Hadoop and Spark. Hence, knowing which threads of investigation
Table 4.8: Instruction-Level Properties of Hadoop MapReduce Examples

<table>
<thead>
<tr>
<th>E.no</th>
<th>Example</th>
<th>IC</th>
<th>CFI</th>
<th>Jumps</th>
<th>Calls</th>
<th>Returns</th>
<th>% CFI</th>
<th>% Jumps</th>
<th>% Calls</th>
<th>% Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aggregate word count</td>
<td>81713</td>
<td>17195</td>
<td>12722</td>
<td>4009</td>
<td>464</td>
<td>21.04%</td>
<td>15.57%</td>
<td>4.91%</td>
<td>0.57%</td>
</tr>
<tr>
<td>2</td>
<td>aggregate word hist</td>
<td>48428</td>
<td>9812</td>
<td>7133</td>
<td>2366</td>
<td>313</td>
<td>20.26%</td>
<td>14.73%</td>
<td>4.89%</td>
<td>0.65%</td>
</tr>
<tr>
<td>3</td>
<td>bbp</td>
<td>85514</td>
<td>17880</td>
<td>13182</td>
<td>4211</td>
<td>487</td>
<td>20.91%</td>
<td>15.42%</td>
<td>4.92%</td>
<td>0.57%</td>
</tr>
<tr>
<td>4</td>
<td>dist bbp</td>
<td>68283</td>
<td>13880</td>
<td>10234</td>
<td>3238</td>
<td>408</td>
<td>20.33%</td>
<td>14.99%</td>
<td>4.74%</td>
<td>0.60%</td>
</tr>
<tr>
<td>5</td>
<td>grep</td>
<td>81404</td>
<td>16911</td>
<td>12501</td>
<td>3937</td>
<td>473</td>
<td>20.77%</td>
<td>15.36%</td>
<td>4.84%</td>
<td>0.58%</td>
</tr>
<tr>
<td>6</td>
<td>pi</td>
<td>65397</td>
<td>13607</td>
<td>10170</td>
<td>3070</td>
<td>367</td>
<td>20.81%</td>
<td>15.55%</td>
<td>4.69%</td>
<td>0.56%</td>
</tr>
<tr>
<td>7</td>
<td>random text writer</td>
<td>70909</td>
<td>14896</td>
<td>11186</td>
<td>3332</td>
<td>378</td>
<td>21.01%</td>
<td>15.78%</td>
<td>4.70%</td>
<td>0.53%</td>
</tr>
<tr>
<td>8</td>
<td>random writer</td>
<td>91414</td>
<td>19462</td>
<td>14508</td>
<td>4475</td>
<td>479</td>
<td>21.29%</td>
<td>15.87%</td>
<td>4.90%</td>
<td>0.52%</td>
</tr>
<tr>
<td>9</td>
<td>sort</td>
<td>101298</td>
<td>21420</td>
<td>16003</td>
<td>4885</td>
<td>532</td>
<td>21.15%</td>
<td>15.80%</td>
<td>4.82%</td>
<td>0.53%</td>
</tr>
<tr>
<td>10</td>
<td>tera gen</td>
<td>134747</td>
<td>28228</td>
<td>21013</td>
<td>6516</td>
<td>699</td>
<td>20.95%</td>
<td>15.59%</td>
<td>4.84%</td>
<td>0.52%</td>
</tr>
<tr>
<td>11</td>
<td>tera sort</td>
<td>121541</td>
<td>25420</td>
<td>18925</td>
<td>5827</td>
<td>668</td>
<td>20.91%</td>
<td>15.57%</td>
<td>4.79%</td>
<td>0.55%</td>
</tr>
<tr>
<td>12</td>
<td>tera validate</td>
<td>139583</td>
<td>29244</td>
<td>21838</td>
<td>6630</td>
<td>776</td>
<td>20.95%</td>
<td>15.65%</td>
<td>4.75%</td>
<td>0.56%</td>
</tr>
<tr>
<td>13</td>
<td>word count</td>
<td>77393</td>
<td>16341</td>
<td>12100</td>
<td>3791</td>
<td>450</td>
<td>21.11%</td>
<td>15.63%</td>
<td>4.90%</td>
<td>0.58%</td>
</tr>
<tr>
<td>14</td>
<td>word mean</td>
<td>62412</td>
<td>13093</td>
<td>9726</td>
<td>2994</td>
<td>373</td>
<td>20.98%</td>
<td>15.58%</td>
<td>4.89%</td>
<td>0.60%</td>
</tr>
<tr>
<td>15</td>
<td>word median</td>
<td>66401</td>
<td>13435</td>
<td>9869</td>
<td>3161</td>
<td>405</td>
<td>20.23%</td>
<td>14.86%</td>
<td>4.76%</td>
<td>0.61%</td>
</tr>
<tr>
<td>16</td>
<td>word standard deviation</td>
<td>82079</td>
<td>16917</td>
<td>12492</td>
<td>3932</td>
<td>493</td>
<td>20.61%</td>
<td>15.22%</td>
<td>4.79%</td>
<td>0.60%</td>
</tr>
</tbody>
</table>

Average Values

|          | 86157 | 17984 | 13350 | 4148  | 485   | 20.83%| 15.45% | 4.81% | 0.57% |

...to follow and which to ignore was difficult and challenging. Execution time and code size of the experiments are the measured parameters for this work. The overhead in the experiments is calculated from time measurements. The time taken to detect an attack in a process $p$ is divided by the execution time of the same process and the result is multiplied by 100 to find the percentage of time overhead, as given in equation 4.3. Here $\text{time}_{\text{detect}}(p)$ is calculated using system clock measurements for encrypting process analysis information, decrypting received messages and hash matching. The communication cost in sending data packets from one node to another is not included. The overhead calculations show the worst case scenario since the input arguments are intentionally low for some...
Table 4.9: Instruction-Level Properties of Spark Performance Test ML Algorithms

<table>
<thead>
<tr>
<th>E.no</th>
<th>Algorithm</th>
<th>IC</th>
<th>CFI</th>
<th>Jumps</th>
<th>Calls</th>
<th>Returns</th>
<th>% CFI</th>
<th>% Jumps</th>
<th>% Calls</th>
<th>% Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fp growth</td>
<td>216009</td>
<td>46544</td>
<td>35200</td>
<td>10251</td>
<td>1093</td>
<td>21.55%</td>
<td>16.30%</td>
<td>4.75%</td>
<td>0.51%</td>
</tr>
<tr>
<td>2</td>
<td>word2vec</td>
<td>147737</td>
<td>30235</td>
<td>22638</td>
<td>6772</td>
<td>825</td>
<td>20.47%</td>
<td>15.32%</td>
<td>4.58%</td>
<td>0.56%</td>
</tr>
<tr>
<td>3</td>
<td>chi-sq feature</td>
<td>172014</td>
<td>35783</td>
<td>26736</td>
<td>8119</td>
<td>928</td>
<td>20.80%</td>
<td>15.54%</td>
<td>4.72%</td>
<td>0.54%</td>
</tr>
<tr>
<td>4</td>
<td>spearman</td>
<td>194615</td>
<td>41043</td>
<td>30857</td>
<td>9155</td>
<td>1031</td>
<td>21.09%</td>
<td>15.86%</td>
<td>4.70%</td>
<td>0.53%</td>
</tr>
<tr>
<td>5</td>
<td>pearson</td>
<td>184628</td>
<td>38694</td>
<td>28996</td>
<td>8691</td>
<td>1007</td>
<td>20.96%</td>
<td>15.71%</td>
<td>4.71%</td>
<td>0.55%</td>
</tr>
<tr>
<td>6</td>
<td>block matrix mult</td>
<td>195714</td>
<td>41245</td>
<td>31030</td>
<td>9174</td>
<td>1041</td>
<td>21.07%</td>
<td>15.85%</td>
<td>4.69%</td>
<td>0.53%</td>
</tr>
<tr>
<td>7</td>
<td>summary statistics</td>
<td>196555</td>
<td>41034</td>
<td>30736</td>
<td>9235</td>
<td>1063</td>
<td>20.88%</td>
<td>15.64%</td>
<td>4.70%</td>
<td>0.54%</td>
</tr>
<tr>
<td>8</td>
<td>pca</td>
<td>192280</td>
<td>40427</td>
<td>30377</td>
<td>9020</td>
<td>1030</td>
<td>21.03%</td>
<td>15.80%</td>
<td>4.69%</td>
<td>0.54%</td>
</tr>
<tr>
<td>9</td>
<td>svd</td>
<td>143996</td>
<td>29684</td>
<td>22334</td>
<td>6550</td>
<td>800</td>
<td>20.61%</td>
<td>15.51%</td>
<td>4.55%</td>
<td>0.56%</td>
</tr>
<tr>
<td>10</td>
<td>gmm</td>
<td>170722</td>
<td>35655</td>
<td>26848</td>
<td>7898</td>
<td>909</td>
<td>20.88%</td>
<td>15.73%</td>
<td>4.63%</td>
<td>0.53%</td>
</tr>
<tr>
<td>11</td>
<td>kmeans</td>
<td>170694</td>
<td>35842</td>
<td>26957</td>
<td>7962</td>
<td>923</td>
<td>21.00%</td>
<td>15.79%</td>
<td>4.66%</td>
<td>0.54%</td>
</tr>
<tr>
<td>12</td>
<td>als</td>
<td>181836</td>
<td>38032</td>
<td>28603</td>
<td>8428</td>
<td>1001</td>
<td>20.92%</td>
<td>15.73%</td>
<td>4.63%</td>
<td>0.55%</td>
</tr>
<tr>
<td>13</td>
<td>decision tree</td>
<td>175889</td>
<td>36655</td>
<td>27546</td>
<td>8140</td>
<td>969</td>
<td>20.84%</td>
<td>15.66%</td>
<td>4.63%</td>
<td>0.55%</td>
</tr>
<tr>
<td>14</td>
<td>naive bayes</td>
<td>171945</td>
<td>36053</td>
<td>27036</td>
<td>8082</td>
<td>935</td>
<td>20.97%</td>
<td>15.72%</td>
<td>4.70%</td>
<td>0.54%</td>
</tr>
<tr>
<td>15</td>
<td>glm classification</td>
<td>186454</td>
<td>39088</td>
<td>29362</td>
<td>8715</td>
<td>1011</td>
<td>20.96%</td>
<td>15.75%</td>
<td>4.67%</td>
<td>0.54%</td>
</tr>
<tr>
<td>16</td>
<td>glm regression</td>
<td>200255</td>
<td>42439</td>
<td>32920</td>
<td>9346</td>
<td>1073</td>
<td>21.19%</td>
<td>15.99%</td>
<td>4.67%</td>
<td>0.54%</td>
</tr>
</tbody>
</table>

Average Values: 181334 | 38028  | 28580  | 8471   | 977    | 20.95% | 15.74% | 4.67%   | 0.54%     |

of the experiments. Real-world big data programs will be much more complex jobs and hence the overhead will be much lesser than what is shown here. Tables 6 and 7 and Figures 4.8a and 4.8b show the analysis of run-times for executing the experiments and the model built from the proposed system. On average, the overhead of running the model is 3.28%. Linear regression and best-fit plots, given in Figures 4.9a and 4.9b, were used to show the relation between programs (given in number of control flow instructions of their assembly representations) and time to detect an attack in them. The time taken to execute example number 4 i.e. distributed bbp program of Hadoop MapReduce example set was too high (288 seconds) to plot on the graph shown in Figure 4.8a.
Figure 4.7: Consistent Distribution of CFI in Hadoop and Spark Tests

(a) Hadoop MapReduce Examples

(b) Spark-Perf Machine Learning Tests
Table 4.10: Run Time Analysis to Analyze and Compare Hadoop MapReduce Examples

<table>
<thead>
<tr>
<th>Exp.no</th>
<th>Time to Execute</th>
<th>Time to Detect</th>
<th>% Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>17.56</td>
<td>0.69</td>
<td>3.93%</td>
</tr>
<tr>
<td>02</td>
<td>20.14</td>
<td>0.42</td>
<td>2.10%</td>
</tr>
<tr>
<td>03</td>
<td>6.39</td>
<td>0.76</td>
<td>11.84%</td>
</tr>
<tr>
<td>04</td>
<td>287.62</td>
<td>0.67</td>
<td>0.23%</td>
</tr>
<tr>
<td>05</td>
<td>7.96</td>
<td>0.79</td>
<td>9.89%</td>
</tr>
<tr>
<td>06</td>
<td>6.48</td>
<td>0.72</td>
<td>11.12%</td>
</tr>
<tr>
<td>07</td>
<td>37.63</td>
<td>0.77</td>
<td>2.05%</td>
</tr>
<tr>
<td>08</td>
<td>31.51</td>
<td>0.97</td>
<td>3.07%</td>
</tr>
<tr>
<td>09</td>
<td>41.71</td>
<td>1.57</td>
<td>3.75%</td>
</tr>
<tr>
<td>10</td>
<td>4.45</td>
<td>1.46</td>
<td>32.82%</td>
</tr>
<tr>
<td>11</td>
<td>4.99</td>
<td>1.37</td>
<td>27.37%</td>
</tr>
<tr>
<td>12</td>
<td>4.61</td>
<td>1.47</td>
<td>31.96%</td>
</tr>
<tr>
<td>13</td>
<td>6.68</td>
<td>0.99</td>
<td>14.86%</td>
</tr>
<tr>
<td>14</td>
<td>6.63</td>
<td>0.90</td>
<td>13.63%</td>
</tr>
<tr>
<td>15</td>
<td>6.64</td>
<td>0.92</td>
<td>13.82%</td>
</tr>
<tr>
<td>16</td>
<td>7.76</td>
<td>1.08</td>
<td>13.88%</td>
</tr>
</tbody>
</table>

Average Values: 31.17, 0.97, 3.12%

\[
\%_{\text{overhead}}(p) = \frac{\text{time}_{\text{detect}}(p)}{\text{time}_{\text{execute}}(p)} \times 100 \tag{4.3}
\]

The proposed system performs a similarity check of control flow within duplicate processes running on different nodes of a big data cluster. This control flow similarity check is performed by matching control instruction sequences. Since the infected node is predetermined in the experiments, the test cases do not have a false positive or false negative. But a false positive will occur when all data nodes are attacked in the same way. A false negative will occur in case of runtime attacks or attacks that originate outside the big data platform. But given the attack model, such cases are not in the scope of this work. Instead, the control flow of the programs is studied to check for vulnerabilities within the programs. Results from tables 4.8 and 4.9 and figures 4.7a and 4.7b show instruction level properties of the examples and tests used in the experiments. It can be observed
Table 4.11: Run Time Analysis to Analyze and Compare Spark-perf MLlib Tests

<table>
<thead>
<tr>
<th>Exp.no</th>
<th>Time to Execute</th>
<th>Time to Detect</th>
<th>% Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.92</td>
<td>0.34</td>
<td>11.67%</td>
</tr>
<tr>
<td>2</td>
<td>12.942</td>
<td>0.24</td>
<td>1.87%</td>
</tr>
<tr>
<td>3</td>
<td>3.899</td>
<td>0.28</td>
<td>7.19%</td>
</tr>
<tr>
<td>4</td>
<td>15.708</td>
<td>0.33</td>
<td>2.08%</td>
</tr>
<tr>
<td>5</td>
<td>3.314</td>
<td>0.31</td>
<td>9.23%</td>
</tr>
<tr>
<td>6</td>
<td>3.011</td>
<td>0.34</td>
<td>11.31%</td>
</tr>
<tr>
<td>7</td>
<td>5.312</td>
<td>0.35</td>
<td>6.63%</td>
</tr>
<tr>
<td>8</td>
<td>8.124</td>
<td>0.34</td>
<td>4.23%</td>
</tr>
<tr>
<td>9</td>
<td>24.647</td>
<td>0.30</td>
<td>1.21%</td>
</tr>
<tr>
<td>10</td>
<td>4.584</td>
<td>0.33</td>
<td>7.24%</td>
</tr>
<tr>
<td>11</td>
<td>7.529</td>
<td>0.35</td>
<td>4.69%</td>
</tr>
<tr>
<td>12</td>
<td>16.884</td>
<td>0.36</td>
<td>2.12%</td>
</tr>
<tr>
<td>13</td>
<td>31.963</td>
<td>0.37</td>
<td>1.17%</td>
</tr>
<tr>
<td>14</td>
<td>1.664</td>
<td>0.37</td>
<td>22.34%</td>
</tr>
<tr>
<td>15</td>
<td>8.151</td>
<td>0.41</td>
<td>5.05%</td>
</tr>
<tr>
<td>16</td>
<td>8.542</td>
<td>0.45</td>
<td>5.26%</td>
</tr>
<tr>
<td>Average Values</td>
<td>9.950</td>
<td>0.34</td>
<td>3.44%</td>
</tr>
</tbody>
</table>

that only 20.8% of the total instruction count in the Hadoop MapReduce examples account for control flow instructions. In case of spark performance tests for machine learning algorithms, 20.9% of instructions in the assembly code are control flow instructions. Of all control flow instructions, jumps are the most significantly used CFI with a lion share of 15.45% of the total instruction count in Hadoop MapReduce examples and 15.74% of the total instruction count in spark performance tests. Calls and returns cover only 4.8% and 0.5% respectively in the Hadoop MapReduce example set and; 4.6% and 0.5% respectively in the spark performance tests set.

It can be inferred from these results that control flow instructions account for only one-fifth of the total instruction count for a program (assembly code). This is a remarkable coincidence among these two sets of programs because (a) they belong to different domains - MapReduce on Hadoop, machine learning in Spark; (b) their source programming language is different - java for Hadoop MapReduce examples, scala for spark-perf machine learning tests; and (c) they differ in program
Figure 4.8: Run-time Analysis of Hadoop and Spark Programs
Figure 4.9: Forecast Plots (Best-fit Regression) - Time to Detect an Attack

(a) Detection Algorithm on Hadoop Cluster

(b) Detection Algorithm on Spark Cluster
size - 86,000 instructions on average per program for the MapReduce example set and 180,000 instructions on average per program for the spark perf machine learning tests. This observation strengthens the initial argument that generating dynamic CFG for large and complex big data programs is cumbersome. This is because the size of CFG is proportional to the code lines which is related to the number of instructions. Hence, the proposed idea of generating CIS and hashing them is a good alternative to the CFG memory complexity problem. The overhead incurred in using the model built from the proposed system architecture is less than 3.28% if it is hosted by the same hardware that hosts the big data systems. This is in the acceptable range of overhead for big data platforms such as Hadoop. The time taken by the proposed system to analyze the programs and compare the results is linearly dependent on the number of control flow instructions in the program, but not on the number of lines of assembly code. This greatly reduces the complexity of the similarity analysis from the conventional and complex approach of generating a CFG. Also, generating CIS only needs a one time parse through the program code (assembly code) and can be performed independently and in parallel on each node of the cluster. The experimental results show the feasibility of implementing a model of the proposed system. Building and implementing a detailed version of this system will demonstrate lower overhead and convince the vendors to adopt it.
CHAPTER 5 : RUN-TIME TECHNIQUES\footnote{Portions of this chapter were previously published in Santosh Aditham, Nagarajan Ranganathan, and Srinivas Katkoori. Memory access pattern based insider threat detection in big data systems. In Big Data (Big Data), 2016 IEEE International Conference on, pages 3625-3628. IEEE, 2016. Permission is included in Appendix A.}

Having compile-time security techniques will help in reducing the scope of attacks. The next set of security techniques proposed in this work are dynamic run-time driven methods. While the compile-time techniques focused on control-flow analysis, the run-time techniques focus on memory behavior analysis of programs. A run-time security technique is proposed to detect an intrusion after it attacks the big data system and another security technique is proposed to predict the possibility of an intrusion before it can attack the system. It makes more sense to have a threat model for the run-time security techniques because the focus is on protecting the big data system in general by dynamically observing system behavior instead of protecting the system from specific attacks. These run-time techniques are add-ons to the compile-time techniques that are more focused on attack detection.

5.1 Threat Model

A software-centric threat model is used in this work that dissects the big data platform design to understand its operational vulnerabilities. The main threats considered in this work are related to data degradation, data theft and configuration manipulation. These threats are a subset of insider attacks that are not typically handled in the big data world. The common link in these threats is that they are directly related to compromising the data hosted by the cluster. Hence,
they have an obvious relation with the cluster infrastructure and memory behavior of datanodes. The threat model does not consider vulnerabilities originated outside the cluster such as user and network level vulnerabilities.

The primary focus of when detecting threats is to mitigate the effect of operational vulnerabilities caused at runtime due to improper usage of system and library calls by programmers. The reason for this choice is two-fold: (a) impact of operational vulnerabilities due to programmer errors cannot be estimated upfront, and (b) programmer errors are usually considered to be resolved at compile time. The threat model also includes illegitimate use of data access privileges by insiders. For example, the proposed system should identify rogue datanodes masquerading as normal datanodes. This can be achieved by analyzing memory access patterns at process level. But it is difficult to differentiate between unusual accesses and corrupt accesses. For the scope of this work, both scenarios are treated as threats. Some assumptions about the system were made to fit this threat model.

- All datanodes use the same architecture, operating system and page size.
- Replica nodes host similar data which is not necessarily the case in a production environment.
- The path to framework installation is the same on all datanodes.
- The communication network will always be intact.
- The communication cost among replicas (slaves) is at most the communication cost between namenode (master) and datanodes (slaves).
5.2 Technique Based on Memory Access Patterns and System Calls

One of the popular ways of detecting intrusions is with the help of pattern matching, which has been around for almost three decades [151]. Typically in pattern matching, there is a defined class of patterns representing good or expected behavior and raw data is matched against this class to identify anomalies. Some of the common parameters used in pattern matching for intrusion detection are: system calls, user activity, network packets, events. For example, using system information such as system calls for intrusion detection is quite popular in the security domain for the past two decades [68, 69]. In this work, the idea of pattern matching for intrusion detection is borrowed and two system parameters are used for adopting this in the big data security domain: (a) system calls (b) memory accesses.

5.2.1 Proposed Methodology

In this work, the idea of building a process behavior profile based on its system call frequencies and memory access patterns is proposed. Three features of memory access are used to describe the memory access pattern: resident set, shared pages, private pages. Once a process behavior profile is built locally, it is then compared with the profile of the process that is scheduled to do the same job on a replica datanode.

The first part of the proposed solution deals with building a behavior profile for every process running on a datanode. Process behavior profiles can be built by observing a variety of process characteristics. For this work, the behavior of a process is described based on the system & library calls and the memory accesses it makes during its runtime. While system and library calls help understand the work done by a process, memory accesses talk about the data usage of a
This makes it hard for the attackers to masquerade. The algorithm for creating a process behavior profile is given in Algorithm 7. The process behavior profile representing the datanode will be a data structure with three entries: (1) identifier, (2) map with one entry per call, and (3) $t^2$, T-squared vector from PCA on memory access information. The identifier needs to be similar for a process across datanodes.

A call instruction in a program has the potential to take the control away from program space to unknown territories. This property of a call instruction makes it an attractive target for attackers. In this work, the focus is on two specific kinds of call instructions known as the system calls and library calls. The system calls are programmatic ways of requesting a kernel service from the operating system. The list of possible system calls is specific to the operating system in use and the number of possible system calls is usually constant and limited. For example, Linux family of operating systems have approximately 140 system calls [152]. Since big data platforms are the target for this work, it is implicit that a certain framework such as hadoop or spark is installed on the cluster under surveillance. These frameworks use a lot of third party libraries and hence we include library call monitoring as well. The advantage of library call monitoring is that the number of jars and shared library objects can be predetermined and these frameworks should have a predefined installation path which will not change unless there is a system level modification.

The problem with tracing system & library calls made by a process at runtime is that the order in which these calls are made might not persist across multiple runs of the same program. But this is important to the security framework since it tries to match information across replica datanodes. An exact match on the call stack will not work if call information is to be used for intrusion detection in a distributed computing domain. To combat this problem, the process be-
behavior profile is designed to be descriptive of the calls made by a process. Instead of using the call stack, metadata about system & library calls is extracted from the call stack and used for intrusion detection. Each row in a process behavior profile representing a library or a system call describes it using four fields: (a) full class name of the callee, (b) method signature, and (c) source code line number and (d) count of the number of times this call was made by the process. A hash of the full class name is used as index for quick look-up. The other difficulty in using call information for intrusion detection is that the number of calls made by a process does not have to be the same for different datanodes. But a huge variation in the number of times a particular call is made can be used as an indicator for intrusion.

While monitoring system & library calls helps in profiling a process and detecting some attacks, this approach is still susceptible to insider attacks. For example, a rogue datanode can masquerade its identity and send the process information to the security framework before an attack is launched. This will lead to a false negative scenario where the datanodes reach to a consensus about a process even though a rogue node compromised a process. Also, system calls in call stack give relevant information about data used by a process only until a file or device gets mapped to memory. All further read() and write() calls are made on the mapped memory using pointers. Hence, it is important to have an alternate perspective about the process when creating its behavior profile. For this reason, we include memory access information of a process in the behavior profile as well. Memory access information helps in the fine granularity of event reconstruction. Memory accesses during runtime give information about program characteristics such as the size of private and shared memory accessed by the program, number of clean and dirty pages in program memory etc.
Algorithm 7 Algorithm to create process behavior profile

1: procedure Behavior Profile
2: pid ← get the process id of datanode
3: interval ← set periodic interval for measurement
4: getProfile(pid):
5: Profile ← empty map
6: Calls ← call getCalls(pid)
7: MemAccess ← call getMemAccess(pid)
8: Hash ← hash of all call paths
9: Profile ← insert([Hash, Calls], MemAccess)
10: return Profile

11: getCalls(pid):
12: while callstack(pid) = system or library call do
13: callee ← store the callee
14: signature ← store the signature of the method
15: callPath ← store the path
16: callCount ← +1
17: hash ← hash of the path
18: info ← callee, signature, path, count
19: return map(hash, info)

11: getMemAccess(pid):
20: while elapsed=interval do
21: if smaps(j).type = private or shared then
22: thisAccess[0] ← smaps(j).Rss
25: MemAccess ← add thisAccess
26: Result ← call PCA(MemAccess)
27: return Result

There are many advantages of using memory access patterns in behavior profiles, such as:

- information can be gathered periodically.

- can be accomplished easily with hardware support.

- gives insight about the data aspects of a process.

- maintains differential privacy.
Figure 5.1: Memory Behavior of Datanodes in a Hadoop Cluster

Today, most distributed systems are a cluster of nodes in their abstract forms i.e. each node is a virtual machine or a process running on a virtual machine. Hence, the process behavior profile is designed to include memory accesses made by the processes. The downside of this approach, especially with modern operating systems such as Linux, is that memory analysis becomes a complicated topic. It is extremely difficult to know about how memory is organized inside a running process and how the kernel handles the different allocation methods and process forking. For example, most modern operating systems use copy-on-write semantics when forking a process where child process address space is mapped to the same backing pages (RAM) as the parent, except that when the child attempts to write to one of those pages, the kernel transparently copies the memory contents to a new, dedicated page, before carrying out the write. This approach speeds up the forking procedure but complicates the memory analysis.

Usually, the kernel delays the actual allocation of physical memory until the time of the first access. Hence, knowing the actual size of physical memory used by a process (known as resident memory of the process) is only known to the kernel. This memory mapping information from the kernel can be used to analyze the memory pattern of processes. A mapping is a range of contiguous
pages having the same back-end (anonymous or file) and the same access modes. For this work, features of memory mapping that are relatively straightforward to analyze were chosen. The private and shared pages of a process in the RAM are observed as parts of memory access patterns. Private memory always belongs just to the process being observed while shared memory may be shared with its parent and/or children processes. In theory, the number of pages allocated to a process should be equal to the sum of its shared and private pages. To alleviate the penalty of constant monitoring, this information is gathered periodically for every 2 seconds. This time interval is randomly chosen because it is not an important detail for security research.

Two simple and typical big data work flows are used to demonstrate the insights provided by system calls and memory accesses of a process. The first example is about writing a 3GB file to HDFS in a Hadoop cluster. Figure 5.1a shows the results of principal component analysis on memory mappings ($t^2$) of the datanodes. The 3 dimension plot in Figure 5.1b shows orthogonal regression among principal components which are calculated from observations made from memory mapping sizes of a datanode. The three dimensions used in this graph are the three different measurements taken at process level - resident set, private pages and shared pages. The red line indicates that the pages in RAM for a process are a combination of its private and shared pages. One observation or data point that seems to be an outlier. This can be due to multiple reasons such as swapping or giving away allocated memory to other processes in need etc. Table 5.1 has the results of f-test performed on $t^2$ statistic calculated as a result of PCA on the sample memory observations made during this test. A random sample of the smallest and largest memory accesses are taken into consideration for this test. Though this is atypical for statistical tests, the intent of this example is to show that the null hypothesis holds true. The first row in the table ($h = 0$)
Table 5.1: F-Test Results on $t^2$ Statistic when Datanodes are Idle

<table>
<thead>
<tr>
<th>F-Test</th>
<th>Nodes 1 &amp; 2</th>
<th>Nodes 1 &amp; 3</th>
<th>Nodes 2 &amp; 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>0.66</td>
<td>0.74</td>
<td>0.9</td>
</tr>
</tbody>
</table>

indicates the test accepts the null hypothesis. Here the null hypothesis is that the data comes from normal distributions with the same variance. The alternative hypothesis is that the population variance in memory access sizes of a datanode is greater than that of the other datanode. The second row of the table, $p$ values, are very high ($>0.5$) and imply confidence on the validity of the null hypothesis that variance in memory access is equal for all datanodes.

The second example shows the simplest case with a Hadoop cluster. Figure 5.1c gives an insight to the system and library calls made by datanodes when they are idle i.e., no user submitted jobs but just maintenance. Each slave node made a total of 275 system calls during the time of observation. The calls and their frequencies were consistent across all data nodes. This can be observed with the overlapping call frequency patterns in Figure 5.1c. It can be concluded that the datanodes are in harmony by evaluating either sets of information: (a) the call information of idle nodes and, (b) the memory access information while putting a file in HDFS.

Representing memory access pattern as a sequence of access size and using approximate string comparison or edit distance is one way to measure similarity between patterns. But there are many aspects to a memory access and creating a memory access profile with fine grained detail preserves more information for comparison across different machines. A straightforward comparison of all observed memory features is redundant and not always feasible. Hence, an approximation method on multiple features is used when creating and comparing a process profile than using just one feature. In this work, each access pattern includes information about three features of a process.
memory access: (a) size of resident pages in the RAM for the mapping, (b) size of shared pages in that mapping, and (c) size of private pages in the same mapping. PCA is used to fit the measured 3 dimensional data as linear regression and share the resultant $t^2$ information for comparing and verifying memory access patterns of two datanodes. PCA calculates three metrics for a given sample dataset: **coefficients, scores and mean.** The coefficients of a principal component are given in descending order of component variance calculated by the singular value decomposition (SVD) algorithm [119]. This is calculated using Equation 5.1 where $X$ is a principal component, $\lambda$ is the eigenvalue and $Y$ is the eigenvector. The sample means, $\bar{x}$ with $n$ observed memory access sizes per process is calculated using Equation 5.2 where $x_i$ is an individual memory access size from datanode $x$. The sample variances, $\sigma^2_x$ is calculated using Equation 5.3 with $n - 1$ degrees of freedom. Since the measurements use multiple features of a memory access, covariances are the eigenvalues of the covariance matrix of input data and they can be calculated using Equation 5.4 where $W_{x1,x2}$ is the covariance in two features of memory access of datanodes $x$. Here $x_{i,1}$ is the memory access size of the $i^{th}$ observation for the first memory feature. When observing $k$ memory features, there will be an array of values [$x_{i,1}, x_{i,2}...x_{i,k}$] for each observation. Scores are the representations of the input data in the principal component space. The $t^2$ values can be calculated from the memory patterns on each datanode using Equation 5.5. But using PCA, the $t^2$ values are calculated as sum of squares distance from the center of the transformed space. Upon having the $t^2$ values, difference between them can be calculated using one way analysis of variance as given in Equation 5.6 where the null hypothesis is that all group means are equal. Here, $t^2_x$ is the t-squared vector for datanode $x$, $t^2_y$ is the t-squared vector for datanode $y$ and $p$ is the probability that the means of $t^2_x$ and $t^2_y$ are the same.
Algorithm 8 Algorithm to verify process behavior profile

1: procedure Verify Profile
2: \( \text{pid} \leftarrow \text{get the process id from datanode} \)
3: \( \text{Local} \leftarrow \text{behavior profile from this node} \)
4: \( \text{Recv} \leftarrow \text{behavior profiles from other nodes} \)
5: \( \text{compare}() \):
6: \( \text{for thread} \ t \ \text{in} \ \text{pid} \ \text{do} \)
7: \( \text{result1} \leftarrow \text{call} \ \text{CompareCalls}(t) \)
8: \( \text{result2} \leftarrow \text{call} \ \text{CompareMemAccess}(p) \)
9: \( \text{result} \leftarrow \text{result1} \& \text{result2} \)
10: \( \text{notify} \ \text{result} \) \( \triangleright \) similarity in calls & memory accesses
11: \( \text{CompareCalls}(t) \):
12: \( \text{for call} \ c \ \text{in} \ t \ \text{do} \)
13: \( \quad \text{if hash}(c_{path}) = \text{Recv.find()} \ \text{then} \)
14: \( \quad \quad \text{if count}(c_{Local} \ll \gg \text{count}(c_{Recv}) \ \text{then} \)
15: \( \quad \quad \quad \text{return} \ \text{true} \)
16: \( \quad \quad \text{else} \)
17: \( \quad \quad \quad \text{return} \ \text{false} \)
18: \( \text{CompareMemAccess}(p) \):
19: \( \quad \text{if compare}(t^2_{\text{Recv}}, t^2_{\text{Local}}) \ \text{then} \)
20: \( \quad \quad \text{return} \ \text{true} \)
21: \( \quad \text{else} \)
22: \( \quad \quad \text{return} \ \text{false} \)

\[
X = \lambda^{1/2}Y \quad (5.1)
\]
\[
\bar{x} = \frac{\sum_{i=1}^{n_1} x_i}{n_1} \quad (5.2)
\]
\[
\sigma^2_x = \frac{\sum_{i=1}^{n} (\bar{x} - x_i)}{n - 1} \quad (5.3)
\]
\[
W_{x1,x2} = \frac{1}{n - 1} \sum_{i=1}^{n} (x_{i,1} - \bar{x})(x_{i,2} - \bar{x})^T \quad (5.4)
\]
\[
t^2_x = n(\bar{x} - \mu)^T W_x^{-1}(\bar{x} - \mu) \quad (5.5)
\]
\[
p = \text{anova}(t^2_x, t^2_y) \quad (5.6)
\]
Algorithm 9 Algorithm to compare process behavior profiles

1: procedure Compare Profiles
2: \( t^2_{Local} \leftarrow \) get the process profile from datanode
3: \( t^2_{Recv} \leftarrow \) received process profiles
4: for all \( t^2_i \) do
5: \( \text{filter}(t^2_i) \) \hspace{1cm} \( \triangleright \) remove tailing \( t^2 \) values
6: \( \text{sort}(t^2_i) \)
7: if Anova\((t^2_{Local}, t^2_{Recv})\) then
8: compromised \( \leftarrow \) Tukey\((t^2_{Local}, t^2_{Recv})\)
9: return true
10: else
11: return false

The dynamic verifier function is a part of the replica datanodes. It is used to parse a received behavior profile and use the extracted information to verify a local process. It will help in identifying process-level anomalies between two replica datanodes. Two algorithms were proposed as part of anomaly detection: (1) Algorithm 8 is the generic verification algorithm that indicates an anomaly among process behavior profiles, and (2) Algorithm 9 is the comparison algorithm for differentiating between two or more memory access patterns. The system & library calls information is given in a hash map data structure with call as the id and call path as the value. Finding differences at call path level is simple because the lookup() function on the map will return the path in constant time.

For every call made locally by a datanode, the call path is hashed using SHA-1 hashing algorithm and the hash map of calls received from the replica datanodes is looked up for the same hash in its index set. This lookup is quick and a mismatch (or) lack of match indicates that the datanodes used different set of system or library calls to perform the same task. This is a necessary but not a sufficient condition to indicate an intrusion. The additional information about calls available in the behavior profile helps in solidifying the attack detection process. The difference in the number of times a system or library call is called to perform the same task should be less than a predefined threshold, \( \delta \), when comparing processes from different datanodes.
Memory pattern of a process is represented using $t^2$ values of PCA. Since the $t^2$ values follow F-distribution, a comparison among memory patterns can be performed in two steps: (a) by running ANOVA test on the $t^2$ vectors to check if the patterns are different, and (b) by running Tukey test on the results from the ANOVA test to find the attacked datanode. This can also be accomplished by any other tests that assess the equality of variances such as F-test, Levene’s test or Bartlett’s test. In case of ANOVA, if the p-value is low ($< 0.05$) then it confirms the rejection of the null hypothesis with strong statistical significance. Then, a multiple comparison test such as a Tukey test is used to check if the difference in the means of the $t^2$ values is significant. One big shortcoming of this approach is that it does not help in distinguishing between unusual process behavior from corrupt behavior. To be able to overcome such shortcomings, techniques such as reinforcement learning need to be used which is left for future work.
The proposed intrusion detection algorithm needs a strong framework to support it. For this purpose, the proposed security framework can be reused for compile-time intrusion detection in big data platforms [19, 31]. Figure 5.2 shows a high level design of the framework. This framework is equipped with an inter-node, key-based secure communication protocol. All the messages among the datanodes are encrypted and use this communication protocol. The framework is assumed to be hosted on a coprocessor that communicates with the CPU for receiving the input data. Though a detailed design for such a coprocessor is not given, an ASIC based design would be a good idea for such a coprocessor. It is assumed that the communication between the coprocessor and the main processor uses a secure protocol such as the one used by Apple processors to communicate with the secure enclave coprocessor [153]. Adding new security instructions to the instruction set of a regular processor can also suffice. The other two elements of this framework are the process profiling phase and verification & consensus phase. Algorithms 7, 8 and 9 are hosted and used for this purpose. The distributed nature of the algorithms help in conducting the profiling phase and the verification phase independently at each datanode in the cluster. This helps a lot in reducing the time taken for intrusion detection. Attack notification is sent from the primary datanode to the master node when there is a consensus among the datanodes about the existence of an attack. This consensus can be established using one of the popular leader election algorithms or consensus algorithms such as raft and paxos [130].

5.2.2 Experimental Results

To test the proposed solution, a small Amazon EC2 cluster was set-up with 3 datanodes, 1 Namenode and 1 Secondary Namenode. Replication factor of the cluster is set to 3 (default). EC2 m4.xlarge instances were used for hosting the cluster. Each node was running Ubuntu 14.04 and
was equipped with a 2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processor, 4 virtual cores and 16 GB of memory.

In order to simulate a compromised cluster, one of the datanodes was explicitly programmed as the corrupt datanode. This was achieved by using two synthetic intrusions given in Table 5.2. These synthetic intrusions represent different kinds of insider attacks such as: (a) misusing the system access privilege and modifying the system configuration, (b) misusing the data access privilege and copying user data for personal benefits, and (c) misusing the data access privilege and sharing or deleting sensitive user data as revenge against the system. Four of the sixteen hadoop examples that come by default with hadoop installation were used for demonstrating the results. A list of the MapReduce examples used along with a brief description is given in Table 5.3. Tests are conducted by running the Hadoop MapReduce examples one at a time on the cluster. Observations from each data node are logged periodically (every 2 seconds) and later analyzed using the proposed framework. Statistical analysis and graphs were generated using Matlab software [154].

Two aspects of a process - system & library calls and memory accesses are observed while running the Hadoop MapReduce examples on the cluster. The call stack of the process running on the data nodes is monitored. For library & system call information, the path of the concerned jar file or shared library is used. For memory access pattern of a process, the memory footprint and memory consumption of that process are required. Memory footprint is obtained by observing the number of pages referenced by the process. Memory consumption is calculated by looking at the the size of the mapping that is currently resident in RAM and the size of memory currently marked as referenced or accessed. In this work, information available through smaps is used which only reports about memory pages that are actually in RAM. The memory consumption of datanode
Table 5.2: Two Synthetic Intrusions for Testing the Proposed Solution

<table>
<thead>
<tr>
<th>Synthetic Intrusion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify the configuration</td>
<td>Change the configuration on one of the datanodes. For example, allocate less heap space to slowdown process execution.</td>
</tr>
<tr>
<td>Copy and share data</td>
<td>Access HDFS using a script and make unauthorized personal copies. Share the data using third party service such as mail client.</td>
</tr>
</tbody>
</table>

Table 5.3: List of Hadoop MapReduce Examples

<table>
<thead>
<tr>
<th>Exp. Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random text writer</td>
<td>A map/reduce program that writes 10GB of random textual data per node.</td>
</tr>
<tr>
<td>Aggregate word count</td>
<td>An Aggregate based map/reduce program that counts the words in the input files.</td>
</tr>
<tr>
<td>Teragen</td>
<td>Generate one terabyte of randomly distributed data.</td>
</tr>
<tr>
<td>Terasort</td>
<td>Sort one terabyte of randomly distributed data.</td>
</tr>
</tbody>
</table>

Processes are monitored by reading the values from `smaps` of all processes or tasks running on the datanode. There is a series of lines in the `smaps` file of a process for each mapping such as the following: `Size, Rss, Pss, Shared Clean, Shared Dirty, Private Clean, Private Dirty, Referenced, Anonymous, KernelPageSize, MMUPageSize, Locked`. For proof of concept, three such features were picked: `Rss, Shared (clean and dirty), Private (clean and dirty)` because in theory Rss should sum up to the combined value of shares and private.

A common problem for big data related academic researchers is the relative lack of high-quality intrusion detection data sets [155]. This is a much bigger problem if the attacks under consideration are not network related. Hence, in-house synthetic attacks were created. Once the system was setup, two synthetic insider attacks were performed on system while it was executing the four Hadoop MapReduce examples to emulate normal usage of the services.

This attack involves exploitation of access privileges by an insider who is legally allowed to access the system and its configuration files. An insider who is a system admin can modify
the configuration properties of a datanode to intentionally impact the performance of the overall system. To implement this attack, the system admin changes the datanode configuration through the `hdfs-site.xml` file on of the datanodes of the Hadoop cluster. The amount of memory allocated for non-DFS purposes on the datanode were increased by 25% and the number of server threads for the datanode were reduced by changing the handler count to 2. Since this is a one-time modification made by an authorized user whose job entails modification of the configuration files, usual user-profiling will not help in detecting the attack.

This attack involves two cases: (a) the use of non-certified (and untrusted) equipment to transfer data from one machine to another, and (b) the use of certified and trusted software (such as a mail client) to transfer data from one machine to another. Similar to the previous attack, the first step involved in this attack is for the system admin to modify the configuration through the `hdfs-site.xml` file on of the datanodes of the Hadoop cluster. A new location local to the system admin account is added to the DFS data directory property. As a result, all blocks at this datanode have two copies - one copy in the actual HDFS location used while setting up the cluster and another duplicate copy in the system admin's local folder. Next, a script is used to simulate an insider periodically transferring these duplicates files from his local folder of to another remote location using the mail client service or USB device. Since it is not possible for us to connect a USB device to Amazon EC2 instances, the system calls involved with using such a device were included in the attack script.

The results of the Hadoop MapReduce examples are given in Table 5.4. Terasort and Teragen examples were run on a terabyte of data while Random text writer and aggregate word counter used a little more than 10GB of data. Because of this variation in data size, it can be noticed that the
time taken to complete these examples also changed accordingly. To generate the terabyte of input data, Teragen took 109 seconds while Terasort took more than 6 times that amount (695 seconds) to sort the terabyte of data. Random text writer took 22.5 seconds to generate random words of size 10GB and Aggregate word count took just 14 seconds to count the words in that 10GB of input data.

While the Hadoop MapReduce examples were executing the way they are supposed to, the proposed security framework performed its analysis on the datanodes that were contributing to the successful execution of those MapReduce examples.
Table 5.4: Memory Properties of Hadoop MapReduce Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Data Size (Bytes)</th>
<th>Time (seconds) No. of observations</th>
<th>Sum of Squares</th>
<th>F Statistic</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teragen</td>
<td>100000000000</td>
<td>109.313</td>
<td>58770 Node1</td>
<td>60114</td>
<td>0.129</td>
</tr>
<tr>
<td>Terasort</td>
<td>100000000000</td>
<td>694.966</td>
<td>118940 Node2</td>
<td>124088</td>
<td>0.256</td>
</tr>
<tr>
<td>Random Text Writer</td>
<td>1102236330</td>
<td>22.543</td>
<td>29681 Node3</td>
<td>31025</td>
<td>0.094</td>
</tr>
<tr>
<td>Aggregate Word Count</td>
<td>1102250820</td>
<td>14.347</td>
<td>29675</td>
<td>31157</td>
<td>0.069</td>
</tr>
</tbody>
</table>

Looking at Attack 1 results (modifying a datanode configuration), it can be noticed from Figures 5.3 and 5.4 that the Amazon EC2 cluster monitoring metrics such as CPU utilization, Network traffic (bytes in and out) were unable to detect the insider attack while running the Terasort and Teragen examples. But the results from the proposed method for the same Hadoop examples clearly indicate that there is an intrusion in the system, as noticed in Figures 5.5 and 5.6. ANOVA on the $t^2$ vectors from the datanodes indicates that one of the datanodes has a different distribution compared to the other two. This can be observed in the $p$-value column of Table 5.4. In all four examples, the $p$-value is extremely low and indicates strong rejection of the null hypothesis that the means of the three different distributions are similar. The multiple comparison test proves that the means of these distributions are not equal and that datanode 1 (in blue) is the one that is different from the other two datanodes (in red). Figures 5.5a - d show the results of ANOVA and Figures 5.6a - d show the results of multiple comparison test. Interestingly, the call frequency on all nodes for these examples seemed to follow similar patterns and the number of distinct library calls made by a datanode is always constant. When considering call frequency analysis for threat detection, this attack is an example of false positive. It is the system call frequency that hints at the possibility of an attack. Since the memory size and the number of threads for datanode1 were
Figure 5.5: Analysis (ANOVA) of Cluster Behavior After a Replica was Compromised.

As for Attack 2 results (illicit copying of data), a USB drive cannot be used to copy files as the test setup uses Amazon EC2. Instead, the data was accessed from the `/dev` folder because

reduced and compared to the other two datanodes, it can be noticed that the system calls (calls to the stack) are relatively low for datanode1 in all examples. This can be observed in Figure 5.7.
all nodes in the cluster are running on Linux operating system. It must be noted that for this kind of attack, it is not required to perform an action (run an example) to notice that the system has been compromised. Hence, this analysis is performed when the system is idle. A script used for encrypting and sending files in RAM disks as mail attachments to system admin's personal email
Figure 5.7: Cluster-level Call Frequencies After a Replica was Compromised by Attack 1.

account. Each file is 4MB in size and it is zipped before sending out as mail attachment. This
leads to a difference in the call frequency pattern of the datanode, as observed in Figure 5.8. It
can be observed from the call frequency in Figures 5.8a and 5.8b that compromised datanode i.e.
datanode1’s call frequency is order of magnitude more when compared to datenode2 and datanode3
which were not compromised.
5.2.3 Analysis of Algorithm and Results

The time complexity of PCA is $O(p^2n + p^3)$ where $n$ is the number of observations and $p$ is the number of variables in the original dataset. In this case, $p = 3$ and even in general, the value of $p$ will be some constant $k$ because it represents the number of features in memory to be observed. Also, this constant $k$ will be much smaller than $n$. So, the time complexity of PCA in this case should be approximately $O(n)$ i.e., linearly dependent on the number of observations made.

In case of memory pattern analysis, if the tails in the observed populations have considerably larger values compared to the mean of the non-tail data, then those data points will have an impact on the output of variance analysis tests such as ANOVA. Hence, it is important to first filter out such data points before running the analysis test. In case of call analysis, there cannot be a concrete conclusion about the system being attacked based only on frequency of calls obtained from different datanodes. Hence, we conclude that a combination of both of these methods along with
other traditional security methods is needed to keep the system safe. Typically, intrusion detection methods need to account every aspect in a confusion matrix. As for the proposed method:

- **True Positive:** successful identification of anomalous or malicious behavior. The proposed framework achieves this for all data attacks because accessing data involves memory allocation or reference.

- **True Negative:** successful identification of normal or expected behavior. The proposed framework achieved this when tested on idle datanodes.

- **False Positive:** normal or expected behavior is identified as anomalous or malicious. The proposed framework will have this problem if the memory mapping observations are not properly cleaned (as mentioned above). A false positive in the proposed framework might also occur when there is a delay in the communication among datanodes about the profile.

- **False Negative:** anomalous or malicious behavior should have been identified but the framework could not. This case arises if the all duplicate datanodes in the big data cluster are attacked by an insider at once. Luckily, this is highly unlikely to happen in case of large, distributed big data clusters. Other traditional security mechanisms in place will be able to prevent such cases from happening [8, 9].

### 5.3 Prediction of Attacks Based on Memory Access Patterns

Until now, the proposed security techniques are useful for intrusion detection. These techniques analyze a program both at compile-time and run-time to detect the existence of an attack. This makes it hard for a perpetrator to circumvent the security techniques and complete an attack.
successfully. But with the increasing competency of attackers worldwide and the increasing sensitivity of data hosted by big data systems, it is important to be able to predict attacks as well. Luckily, artificial intelligence is on the rise and new technologies like machine learning and deep learning can be leveraged by the security domain to come up with novel intrusion prediction techniques. In this work, a deep learning technique called Long Short-Term Memory (LSTM) is used for time-series prediction of data attacks in a big data system.

5.3.1 Proposed Methodology

Memory access pattern of a process can be modeled as a time-series where each data point is the size of the memory used by the process at a particular instance. The latest operating systems have advanced memory management techniques where each memory access can be described using multiple features. The memory access pattern of a process is modeled as a multi-featured time series and use LSTM to predict the size of the next memory access. The intention behind this approach is to associate a process to a predefined class of memory usage and verify, during run-time, if the process behavior is in sync with the generalized behavior of processes belonging to its class. A novel method is proposed to fit this idea in the two-step intrusion detection framework. The overall algorithmic flow of the proposed method can be seen in Figure 5.9.

Proposing the use of machine learning techniques for prediction models is not new. For example, Allesandro et al. [156] proposed the use of Support Vector Machines for predicting application failure. Murray et al. [157] used naive Bayesian classifiers for predicting failure in hard drives. Recently, researchers from Facebook [122] used stack-augmented recurrent networks to predict algorithm behavior. We believe that predicting memory behavior of datanodes in a big data cluster to anticipate data oriented insider attacks has not been done before. In this work, we use
LSTM as a tool for prediction but the same can be achieved using other prediction techniques as well.

The first step when trying to predict or detect attacks in a system is to understand the way the system works. When a job is submitted to the big data cluster, a resource manager or scheduler such as yarn will schedule that job to run on datanodes that host the required data. Each replica datanode is supposed to execute the same job on their copy of the data to maintain data consistency. This replication property of a big data cluster helps in predicting an attack. By monitoring the memory behavior of replica datanodes and understanding their correlation, the memory requirement of a datanode in near future can be predicted with some certainty.

The proposed memory behavior prediction method involves running a LSTM network on the memory mapping data obtained locally and independently at each datanode of the big data cluster. For this work, four different features of memory are monitored:

- the actual size of memory mappings.
- Resident set (RSS) that varies depending on caching technique.
- Private pages to characterize a particular workload.
• Shared pages to understand the datanode workload in the background, especially when the job is memory intensive.

Hence, each data point in the observed memory access sequence is a vector of size four. Since the readings are taken at regular time intervals, this data resembles a time series. From Equation 2.1, the input to the LSTM cell at time step \( t - 1 \) is \( i_{t-1} \) which is a data point with four values \(< a, b, c, d >\) where \( a \) is the size of the mapping, \( b \) is the size of pages in the resident set of the memory, \( c \) is the size of private pages and \( d \) is the size of shared pages. The recurrent LSTM network in this case is simply a single LSTM cell with a feedback loop. The decision reached at time step \( t - 1 \) affects the decision it will reach at time step \( t \). After a memory data point is fed to the LSTM cell as input, the output \( o_t \) at the next time step \( t \) is the predicted output. During training phase, this output is compared with the actual observed value for correctness and the weights and bias of the LSTM cell are adjusted accordingly. For this purpose, a standard metric called Root Mean Squared Error (RMSE) is used. RMSE, as given in Equation 5.7, is a general purpose error metric for numerical predictions. Here, \( n \) is the number of observations in the sequence, \( y_k \) represents a single observation and \( \hat{y} \) denotes the mean of the sequence so far. It is sensitive to small errors and severely punishes large errors. When the RMSE falls above/below a predefined threshold \( T \), it is an indication the datanode behavior is abnormal. But an observed anomaly in the memory behavior at a datanode at this stage is only a necessary but not a sufficient condition to predict an attack. For this reason, a second step called consensus is introduced in the proposed method. The data points representing the observed and the expected memory usage for a certain time window \( \tau \) is shared with other replica nodes using the secure communication protocol for consensus.
\[ RMSE = \sqrt{\frac{1}{n} \sum_{k=1}^{n} (y_k - \hat{y})^2} \] (5.7)

The same framework is used to host the proposed attack prediction method. This framework facilitates secure communication among datanodes to exchange their local analysis results, share the statistical test results for consensus and notify the master node, if necessary. The framework uses encryption/decryption techniques such as AES along with communication protocols such as SSH and proposes the use of hardware security such as TPM and secure co-processors for key generation, storage and hashing.

When a datanode receives memory behavior data from other replica datanodes, it compares that data with its local data to verify for similarity. The procedure used for such comparison can vary among different implementations. In this work, the comparison technique uses statistical analysis methods: ANOVA for variance and Tukey for multiple comparison. Both datasets are two-dimensional with predicted values and actual values. Each replica datanode runs a one-way variance test, ANOVA to measure correlation between local data and received data. If the null hypothesis is rejected, then a multiple comparison test (Tukey test) is used to find the corrupt datanode. When a consensus is reached among datanodes about a possible attack, necessary prevention steps such as notifying the master node can be taken.

5.3.2 Experimental Results

This section presents the experiments and their results in two parts: the first part describes the setup of Hadoop cluster, LSTM network and the choice of programs to test the proposed method; the next part explains the results observed from the conducted experiments.
Table 5.5: Machine Configurations

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Hadoop</td>
</tr>
<tr>
<td></td>
<td>LSTM</td>
</tr>
<tr>
<td>EC2 Instance Model</td>
<td>t2.micro</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Xeon with Turbo</td>
</tr>
<tr>
<td></td>
<td>AMD A8 Quad Core</td>
</tr>
<tr>
<td>Compute Units</td>
<td>1 (Burstable)</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>vCPU</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>Memory (GB)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Storage</td>
<td>Elastic Block Store</td>
</tr>
<tr>
<td></td>
<td>SATA</td>
</tr>
<tr>
<td>Networking Performance</td>
<td>low</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux/UNIX</td>
</tr>
<tr>
<td></td>
<td>Ubuntu 16.04</td>
</tr>
<tr>
<td>Software distribution</td>
<td>Hadoop 2.7.1</td>
</tr>
<tr>
<td></td>
<td>Keras</td>
</tr>
<tr>
<td></td>
<td>Tensor Flow</td>
</tr>
</tbody>
</table>

Amazon EC2 is used to set up a small Hadoop cluster of 3 datanodes, 1 namenode and 1 secondary namenode. All programs used for testing are obtained from Hadoop MapReduce examples that come with the basic installation of Hadoop and run on this Hadoop cluster. Some detail about these MapReduce programs are given in Table 5.6. A standard quad-core desktop is used to run Keras & Tensor Flow frameworks required for implementing a LSTM network. Statistical tests were conducted using Matlab. The configuration of the Hadoop nodes and the LSTM machine are given in Table 5.5.

First, each Hadoop MapReduce example is run on the cluster for a hundred times. For every iteration, the memory readings are taken periodically with a time interval of five seconds from /proc/pid/smaps which is a linux tool for observing memory usage of processes. Average values of the four memory features - size, resident set, private pages and shared pages are calculated and stored in an array of four-dimensional vectors. By the end of hundred iterations, a time series history of process behavior is obtained where each data point gives information about the memory behavior of the process for a specific run. This is a part of local analysis and it is done in parallel on every datanode.
Next, the time series data of a process i.e., one hundred time steps of four-dimensional data points, is fed to a LSTM network. The LSTM network has one hidden layer of four LSTM cells and a dense output layer of four neurons (fully connected) to predict four memory features. It looks back at one previous time step when predicting a new value and the batch size is also set to one. 80% of this data is used for training the LSTM network and the remaining 20% of data is used for testing the LSTM network. The number of epochs varied between 50 and 100. RMSE is used to calculate accuracy of the LSTM network for both training and testing phase. By the end of this step, the LSTM network of every datanode is trained and tested on normal behavior of that datanode.

Finally, three different kinds of insider attacks are simulated in the Hadoop cluster and datanode memory behavior when running the Hadoop MapReduce examples is captured again in the same way as mentioned before. The details about these attacks are given in Table 5.7. For demonstration, the datanodes were infected by increasing thread count allocated for HDFS, allowing the datanode to cache and increasing the cache report and cache revocation polling time (making the datanodes faster). Also, the datanodes were running two separate scripts in the background: one for sending emails of data and another to modify logs. This data is then used for testing the LSTM network that has been trained by using the same 80% data from above experiments. The challenge that these attacks impose is that there is no error or change in program behavior with respect to correctness.

There are three sets of results in this work.

- The first set of results show a comparison between the program behavior when the datanodes are not yet compromised and when they are compromised, as shown in Figure 5.10. Here, the
### Table 5.6: List of Hadoop MapReduce Examples

<table>
<thead>
<tr>
<th>E.No.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aggregateword-count</td>
<td>An Aggregate-based map/reduce program that counts the words in the input files.</td>
</tr>
<tr>
<td>2</td>
<td>bbp</td>
<td>A map/reduce program that uses Bailey-Borwein-Plouffe to compute exact digits of Pi.</td>
</tr>
<tr>
<td>3</td>
<td>pentomino</td>
<td>A map/reduce tile laying program to find solutions to pentomino problems.</td>
</tr>
<tr>
<td>4</td>
<td>pi</td>
<td>A map/reduce program that estimates Pi using a quasi-Monte Carlo method.</td>
</tr>
<tr>
<td>5</td>
<td>randomtextwriter</td>
<td>A map/reduce program that writes 10GB of random textual data per node.</td>
</tr>
<tr>
<td>6</td>
<td>randomwriter</td>
<td>A map/reduce program that writes 10GB of random data per node.</td>
</tr>
<tr>
<td>7</td>
<td>sort</td>
<td>A map/reduce program that sorts the data written by the random writer.</td>
</tr>
<tr>
<td>8</td>
<td>wordcount</td>
<td>A map/reduce program that counts the words in the input files.</td>
</tr>
</tbody>
</table>

### Table 5.7: Insider Attacks on a Datanode

<table>
<thead>
<tr>
<th>No</th>
<th>Title</th>
<th>Description</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mail Attack</td>
<td>A system admin is sending out user data as attachments from datanode to personal email account.</td>
<td>Data Privacy</td>
</tr>
<tr>
<td>2</td>
<td>Configuration Attack</td>
<td>A system admin changes the HDFS configuration of a datanode he/she has access to.</td>
<td>Datanode performance</td>
</tr>
<tr>
<td>3</td>
<td>Log Deletion Attack</td>
<td>A system admin modifies user data and deletes the log files on a datanode he/she has access to.</td>
<td>System Analysis</td>
</tr>
</tbody>
</table>
results are shown for one datanode and each Hadoop MapReduce job is analyzed using four different features of memory.

- The next set of results show the accuracy in the prediction of datanode memory behavior by the LSTM network. These results can be observed in Figure 5.11 and Table 5.8. Since the range of the input data is between $(150 - 7500)$, RMSE in the range of $(1 - 60)$ can be considered as good prediction.

- The final set of results show the efficiency of the prediction algorithm when tested on the
Figure 5.11: LSTM Analysis of Memory Behavior of a Datanode for MapReduce Examples
Table 5.8: RMSE Results from LSTM while Predicting Datanodes Memory Mappings

<table>
<thead>
<tr>
<th>E.No.</th>
<th>Epochs</th>
<th>Stage</th>
<th>Samples</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DN1</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>TRAIN</td>
<td>78</td>
<td>7.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(normal)</td>
<td>18</td>
<td>4.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(attack)</td>
<td>57</td>
<td><strong>5586.74</strong></td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>TRAIN</td>
<td>78</td>
<td>7.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(normal)</td>
<td>18</td>
<td>5.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(attack)</td>
<td>57</td>
<td><strong>5747.34</strong></td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>TRAIN</td>
<td>78</td>
<td>6.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(normal)</td>
<td>18</td>
<td>6.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(attack)</td>
<td>37</td>
<td><strong>6007.43</strong></td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>TRAIN</td>
<td>78</td>
<td>11.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(normal)</td>
<td>18</td>
<td>12.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(attack)</td>
<td>22</td>
<td><strong>2063.7</strong></td>
</tr>
<tr>
<td>1+5</td>
<td>50</td>
<td>TRAIN</td>
<td>158</td>
<td>20.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(normal)</td>
<td>38</td>
<td>15.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(attack)</td>
<td>20</td>
<td><strong>6180.51</strong></td>
</tr>
<tr>
<td>6+7</td>
<td>100</td>
<td>TRAIN</td>
<td>238</td>
<td>12.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(normal)</td>
<td>58</td>
<td>11.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TEST(attack)</td>
<td>27</td>
<td><strong>5972.78</strong></td>
</tr>
</tbody>
</table>

Hadoop cluster with corrupt or compromised datanodes. These results can be observed in Figure 5.11 and Table 5.8. Since the range of the input data is between (150 – 20000), RMSE in the range of (1800 – 6700) can be considered as poor prediction.

5.3.3 Analysis of Algorithm and Results

The Hadoop MapReduce examples used in this work represent a variety of workloads. Programs such as Pi, BBP, Pentomino use MapReduce jobs that are compute intensive but use very less memory. The execution time of a single run of these programs is less than 5 seconds. Programs such as Random Write, Sort, Random Text Write, Aggregate Word Count are memory intensive and at the same time they represent multi-tasking. While the Random Write, Random Text Write programs are limited to writing data to HDFS, the Sort, Aggregate Word Count
programs use more memory because they are both memory and compute intensive tasks. Each of these jobs took less than 2 minutes to complete working on their 1GB of input data. The datanode alternates between these two programs for each run. Finally, the Word Count program, which is very similar to Sort, Aggregate Word Count programs shows that the replica datanode behavior is not always similar to one another even if the Hadoop cluster is not under attack. The input for this job was three times that of the other jobs and each job took almost 8 minutes to complete.

While the LSTM network of every datanode is trained using the data from normal runs of the programs, the testing datasets were different. One testing dataset represents normal behavior and the other represents compromised behavior. As shown in Figure 5.12, the convergence in loss plateaus after 50/100 epochs. This shows the efficiency of the LSTM model during training. The values in the test results differ by a huge margin as shown in Test(normal) row and Test(attack)
row of every MapReduce program in Table 5.8. It can be observed that the RMSE increased from a small value in the range of 1-60 to a much bigger value in the range of 1800-6700. That huge difference in the RMSE can be attributed to the type of attacks that were introduced. Since the attacks improved the memory performance of datanodes, they made the datanodes use more memory. Though it is difficult to come up with a fixed number or range in RMSE difference to denote compromised behavior, the idea here is to have application specific threshold, $T$, that represents the acceptable range of deviation between training and testing phase in order to predict an attack successfully. The attacks used in the experiments are designed to explicitly show such huge impact.
CHAPTER 6 : LIMITATIONS AND CONCLUSIONS

In this dissertation, a security framework for big data systems is developed to detect insider attacks quickly with low overhead. The system consists of a two step attack detection algorithm and a secure communication protocol. The first step in our algorithm is to independently conduct local analysis of processes at slave nodes of a big data cluster. The second step is to verify the validity of the analysis from the first step where the replica nodes of the big data cluster come to a consensus about the existence of an attack. The main support for our work comes from the replication property of big data systems because it enforces coherence in program behavior among replica datanodes that is needed to maintain data consistency. The central theme of this work revolves around the idea of delegating security as an independent module for big data systems and hence the various components needed for such security models are proposed and discussed in detail.

First, an intrusion detection technique is introduced that suspects the existence of an attack on nodes with primary copy of data. This is achieved by analyzing the control flow instructions of processes running on them. A custom metric called attack probability score is calculated per processes to help identify attacks in the system and avoid false positives. All communications related to our framework are encrypted using keys generated from master keys unique to their host nodes. Attack confirmations are based on consensus among the replica nodes that host a given dataset. By evaluating our proposed framework on the CPU and Crypto benchmarks of Phoronix Test Suite, it is shown that the proposed can identify attacks with very negligible time overhead (0.01%).
Next, a simple hash string matching technique is proposed to fulfill the distributed process similarity check and identify attacks. A secure communication protocol for data nodes that uses periodically generated random keys is proposed to conduct the detection algorithm. A model of the proposed system is tested in real-time on Amazon’s EC2 clusters using a different sets of Hadoop and Spark programs. The time overhead was 3.28% and it is observed from the results that the proposed security system uses only 20% of program code to detect attacks.

Next, a novel approach that uses control flow analysis to detect program level intrusions is introduced. Behavior of a program is modeled by extracting a MSA set representation of its CFG. Similarity check among duplicate programs is performed by a complete matching among hashed sets of MSAs. Experiments were conducted on real-world Hadoop MapReduce examples and it is observed that the proposed technique takes only 0.8% of execution time to identify intrusions. The naturally sparse nature of CFGs helps in achieving this low overhead.

Then, a runtime technique to mitigate vulnerabilities and detect attacks is proposed. This technique analyzes system & library calls along with memory accesses of a process, packs all of the analysis information together as a process behavior profile and shares that profile with other replica datanodes in the system. The replica datanodes verify the received call traces and access patterns with their local processes for attack detection. Experimental results show that our approach can detect insider attacks even in cases where the usual CPU and network analysis fail to do so, when tested on Hadoop MapReduce examples.

Finally, a method to predict (and successively detect) an internal data attack on a Hadoop cluster is introduced. This technique works by analyzing individual datanode memory behavior using LSTM recurrent neural networks. The core of our idea is to model memory mappings of a
datanode using multiple features and represent that data as time series such that a recurrent neural network such as LSTM can predict program level memory requirements. When the actual memory usage differs from the predicted requirement by a huge margin, datanodes in a big data cluster share that information and come to a consensus regarding the situation. The efficiency of this method is demonstrated by testing it on a Hadoop cluster using a set of MapReduce open-benchmarks and infecting the cluster using predefined attacks. During the course of this work, it is understood that big data systems do not need new security algorithms but they need the old techniques to be applied in new combinations.

6.1 Future Work

For future work, we would like to develop a detailed architecture for coprocessors that can host the proposed framework. Also, there is a lot of scope for improvement with regards to the proposed security techniques:

- The compile-time technique based on attack probability score is based on weak rule engine. So, a well established rule engine that fits in a heterogeneous system with different instruction sets is required.

- The proposed string matching method for CFG has to be compared with other graph isomorphism techniques.

- All control-flow based techniques need to be compared with relevant works such as CFI and symbolic execution.

- More variations of LSTM and other recurrent neural networks need to be tried and tested when predicting attacks.
A major bottleneck for this research was in finding the right set of benchmarks to test our proposed techniques. So, as part of future work, the proposed methods have to be tested and verified on a broader set of industry datasets. Also, all proposed techniques must to be evaluated with security related big data benchmarks when available. All proposed techniques must be compared with other existing security techniques and a formal proof about the efficiency of the techniques proposed is good to have. Finally, we would like to explore the concept of differential privacy.
REFERENCES


APPENDIX A: COPYRIGHT CLEARANCE FORMS

Below is the permission for chapter 4.2 and its Figures and Tables.

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © [Year of original publication].

2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] appear prominently with each reprinted figure and/or table.

3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [Year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication].

2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on line.

3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a license from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.

2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] appear prominently with each reprinted figure and/or table.

3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [Year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication].

2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on line.

3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a license from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

APPENDIX A: COPYRIGHT CLEARANCE FORMS

Below is the permission for chapter 4.2 and its Figures and Tables.

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.

2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] appear prominently with each reprinted figure and/or table.

3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author’s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [Year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication].

2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on line.

3) In placing the thesis on the author’s university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity’s name goes here]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a license from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.

2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] appear prominently with each reprinted figure and/or table.

3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author’s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [Year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication].

2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on line.

3) In placing the thesis on the author’s university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity’s name goes here]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a license from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
Below is the permission for chapter 4.4 and its Figures and Tables.
Below is the permission for chapter 5.2 and its Figures and Tables.