Secdoop: A Confidentiality Service on Hadoop Clusters

by

James Holland Majors

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 9, 2011

Keywords: Hadoop, Cryptography, MapReduce, Distributed Computing

Abstract

The MapReduce model has proven to be an effective way to demonstrate academic research while using distributed file-systems. The MapReduce programming model was introduced by Google in 2004 [1]. MapReduce has proven to be a good solution for large data sets requiring intensive processing. Hadoop, an open-source Java implementation of MapReduce, was created by Yahoo in 2007. Industries that deal with sensitive data in large scales are hesitant to embrace a solution of processing that distributes their sensitive data. Cryptography is often used to protect sensitive data, but it is computing intensive, often making it undesirable as a solution. Utilizing cryptography while distributing the processing over a trusted cluster will improve the overall runtime. This is an excellent solution for large data sets that are sensitive in nature [2][4]. In this paper, we describe two applications that distribute the cryptographic process over a trusted cluster. The first application will handle encryption of an input file that will be placed inside the Distributed File System (DFS). The second application will handle decryption of an input file that is located on the DFS. These two applications will demonstrate the effect of utilizing cryptography while distributing processing over a Hadoop cluster. Our application features the use of cryptography in parallel on a cluster of commodity machines. Our experimental results show the performance increase when dealing with large sets of data.

Acknowledgments

A special thanks to Dr. Xiao Qin for constant colaboration of this research. Thanks to Shu Yin for assistance with the figures created from our collected data. Thanks to Zhiyang Ding for assistance with troubleshooting the RecordReader problems. Thanks to Xiajun Ruan for assistance with troubleshooting cryptography implementations. A special thanks to Gregory Poirier for assistance with troubleshooting, data collection, and the automation of testing. Thanks to Riley Spahn for assistance with validation.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

DFS	Distributed File System
JFS	Java File System
NFS	Network File System
MPI	Message Passing Interface
PVFS	Parallel Virtual File System
JCE	Java Cryptographic Extension
HDFS	Hadoop Distributed File System
AES	Advanced Encryption Standard
DES	Data Encryption Standard
DESede	Triple Data Encryption Standard

Chapter 1

Introduction

MapReduce is a programming model, proposed by Google, based on the Lisp commands "Map" and"Reduce". Implementations of this model are commonly used for processing large data sets [1]. As the size of available storage devices increase in conjuction with the decrease in prices for the devices, it is more cost effective for industries to store larger sets of data. Also, as the complexity of data systems increases, so does the size of data stored by industrial applications. These industries are challenged with the need to efficiently analyze terabytes or even petabytes of information due to the increased data size produced from these applications. There is a need to find a solution that allows for larger amounts of data to be analyzed and the computation to be distributed among multiple machines to increase the processing efficiency. With terabyte hard drives becoming more and more cost effective, businesses can afford to host their own clusters as more open-source products arrive on the scene to assist programmers in their customization efforts. Hadoop is an open-source Java implementation of the MapReduce programming model that can be set up on a wide variety of hardware platforms [2][3][4]. There has been a great deal of research that focuses on the performance of Hadoop. In May of 2009, it was announced that a team at Yahoo! sorted one terabyte of information in 62 seconds [5]. Hadoop has been a proven tool that is used by many of the major companies in the industry, such as: Yahoo!, Amazon, and Facebook, along with many major Universities. The open source availability makes this a common platform for developers to customize applications to solve unique challenges.

The inheritance provided in Java, allows developers to override the Map and Reduce interfaces to tailor the application to their customized needs. The developer can also customize input formats, output formats, and record readers to handle the input splits in an optimal fashion. Application customization is facilitated through the use of Hadoop and Java libraries. Hadoop allows developers an easy way to write parallel programs with minimal code writing. Hadoop configurations are automatically set up to manage the communications between different nodes in the cluster as well as data replication and data storage management.

Once the problem of distribution is addressed through the use of MapReduce, the issue of security arises and is a major consideration in the viability of using distribution as a solution for a company. Whenever there are distributed communications, there is always an issue of privacy that must be considered. Security solutions are very process intensive, but the speed of process distribution opens the door for multiple solutions which have not been feasible in the past. Companies can reduce their security risks through the use of in-house trusted clusters. By having an in-house trusted cluster [6][7][8], the company knows where their data resides in the cloud. Companies can set up firewalls or security policies for their trusted clusters in order to implement whatever security level is warranted for their data. These companies may choose options such as totally blocking user access from outside their network, or limiting access by use of authentication incorporating key management and other security policies.

Chapter 2
Related Work

There have been a multitude of cryptographic file systems that address data confidentiality. The Java File System (JFS) [9] is a distributed solution that works on heterogenious operating systems. The JFS is setup as a client/server architecture and is a reasonable solution when the Network File System(NFS) [10] is not feasible because this solution does not allow for parallel computing over a cluster.

The Message Passing Interface (MPI) is a useful tool when dealing with parallel computing [11][12][13]. MPISec is an implementation of cryptographic services using MPI [14]. MPISec uses the Parallel Virtual File System (PVFS) and does not employ any type of data replication. The lack of data replication is a bottleneck when different nodes in the cluster process segments of the file that arent located locally. Hadoop replicates data over multiple nodes thus eliminatinating the bottleneck, which makes it a good solution. MPI is challenged with managing which processes are allowed access to different parts of the file. The Hadoop namenode, also known as the master, handles the details of access control between map tasks, and Hadoop application developers can rely on the default configuration of the cluster without any extra code added. This reduces complexity for application developers and allows easy implementation of parallel programs.

Biodoop was developed as a framework built on Hadoop because bioinformatics is computationally intense and deals with large data sets [15]. The results from Biodoop's experiments show that Hadoop is a versatile framework for problems with light computation and large data sets, as well as heavy computation with small data sets. This is similar to security in the fact that there are many aspects to security and to be specific, cryptography is very computationally heavy.

SecureMR is a security application that has been implemented using Hadoop [16]. SecureMR focuses on the integrity of the services running MapReduce. The application analyzes the behavior of worker nodes to determine the integrity of the services running. Our solution focuses on data confidentiality rather than service integrity.

The idea of having a cryptographic distributed file system is a sound idea, but it does not fit well with the MapReduce programming model [9]. If cryptography is implemented at the file system level, the overhead would outweigh the benefits of the MapReduce model. This is the main reason that our solution is running at the application level. Our solution is built as a framework that will begin as a cryptographic solution for the MapReduce model and we will continue to build this application into a security suite.

Chapter 3
Motivation And Background

The driving force of this experiment was to increase performance of data processing associated with large datasets by using a distributed system, while utilizing cryptography to meet the security needs. A major factor considered was the cost of the equipment, the open-source availability and the ease of programming using Hadoop. Using this model, the developer creates a custom Map and Reduce function. When the custom application is run, the master node locates the data on the distributed file system and splits up the file into small manageable pieces. Each piece is sent to a worker node(s) and then processed using the Map function. The Map function then produces key/value pairs from the pieces that are then passed back to an intermediate file. The Reduce function receives these intermediate values and combines values that are similar according to the keys. Once the values are combined, the Reduce function returns the result back to the system [1]. The complexity of developing parallel applications is reduced by having only two main functions that are needed in applications for MapReduce. MapReduce has the ability to run on commodity systems that can lower the price of a cluster and makes this option a reasonable solution for companies to have an in-house cluster. The greater scalability provided by the MapReduce programming model allows for the opportunity to expand the model to include more nodes as more processing is needed. This expansion may result in the need to have multiple geographic sites available and the communications between these sites can become a security weak point in the overall system. This vulnerability causes several fields of industry including, but not limited to financial, medical, and military, to lose interest in this solution as sensitive data should not be transmitted in plaintext over unsecured connections. Another option is to use cryptography to protect the sensitive data, but the deterrent lies within the time consuming, intensive processing that is required to use encryption and decryption. One possibility is to use an in-house trusted Hadoop cluster to prepare the data for transmission. Java provides a security library called the Java Cryptographic Extension (JCE) which allows programmers to specify what type of encryption standard and key length is desired using cipher streams. Hadoop can easily employ cryptographic solutions in custom parallel applications using the JCE provided by Java. This allows a low cost and efficient customizable solution to this security vulnerability.

3.1 Hadoop

Hadoop consists of two major parts, the MapReduce model and the Hadoop Distributed File System (HDFS). The HDFS is managed by a "NameNode" that is the master of the cluster. The NameNode manages the file locations, the data replication, and keeps track of storage nodes [2]. The MapReduce portion of Hadoop is managed by the "JobTracker". The JobTracker is responsible for the task scheduling and job monitoring during the MapReduce process. Each node in the cluster contributes both to storage and to computation. The computation portion is handled by the JobTracker, and the storage portion is maintained by the NameNode [2][4]. The NameNode also ensures that each file fragment is replicated across multiple nodes to increase reliability and availability of data. The replication of data also provides a fault tolerance during runtime in case a node is busy or malfunctions. When a job is scheduled that deals with a file that is replicated over multiple nodes, the nodes can begin work immediately on separate parts of the file without consuming the network due to retransmission of file fragments, and this is a result of data replication.

3.2 Secdoop Behavior

Figure 3.1 shows how Secdoop interacts with the NameNode and the TaskTracker of Hadoop. The red objects represent the Hadoop master nodes, and the cluster of worker nodes are represented on the right side of the figure. Secdoop starts by being inserted into

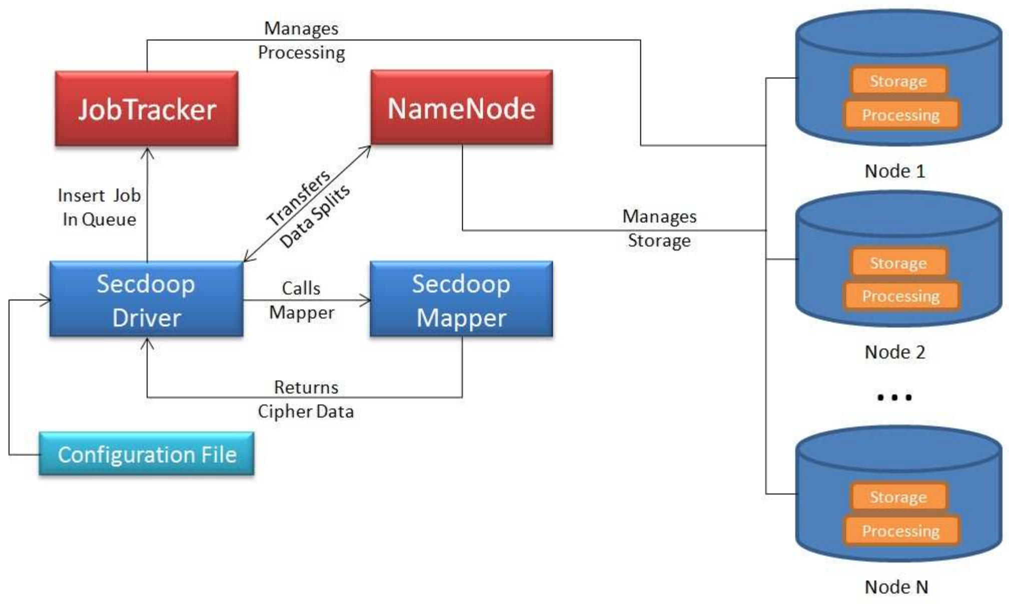

Figure 3.1: Demonstration of How Secdoop Interacts with Hadoop and HDFS

the job queue in the JobTracker. The NameNode then transfers the data splits back to Secdoop. Secdoop then calls the mapper class to process the data from the NameNode. The Mapper then completes and the intermediate data is then returned and then transferred back to the NameNode for storage.

Chapter 4

Research Overview

4.1 Project Description

This research project investigates the use of cryptography in MapReduce with process distribution within a trusted cluster. The run time of three encryption standards will act as a benchmark to demonstrate the performance gain from process distribution. In this project, we have chosen to work with the Advanced Encryption Standard (AES), Data Encryption Standard (DES), and the Triple Data Encryption Standard (DESede). AES employs a variety of bit substitutions, row shifting, column mixing, and different keys for each round of encryption [17]. DES, which is slower than AES, begins with 48 bit keys and iterates through 16 rounds of computations including bitwise XOR operations. DESede is the same as using the DES algorithm three different times with different keys each time [18]. Each of these algorithms is process intensive and is not a good solution for large amounts of data without the distribution of processing. Our experiment will demonstrate the timings for each of the algorithms using the Java Cryptographic Extension library. We have developed two applications to demonstrate the cryptographic process: encrypt and decrypt.

4.2 Assumptions

In our experiments we assume that the application is running on a trusted cluster that is not concerned with data tampering or man-in-the-middle attacks towards communications. For testing purposes we leave key management for future work. The focus on this experiment is to demonstrate the effects of the distribution of processing on the runtime of cryptographic solutions when compared to the processing of a single machine.

4.3 Design Issues

4.3.1 Secdoop Design

Secdoop runs at the application level in order to not interfere with the MapReduce model in Hadoop. If our solution were to run at the file system level, the overhead would defeat the performance increase from the MapReduce model. Since our solution does not protect at the file system level, the plaintext files are needed by the HDFS. The problem with having the plaintext on the HDFS is that if the cluster is compromised, then the encryption is of no use. This is why we suggest our solution to be a trusted in-house cluster that can prepare sensitive data for further distribution. Another issue with file system level cryptography is inheritance in the classes used by Hadoop. Many of the subclasses override super methods and constructors which would cause for a major modification to the entire system to ensure that the cryptographic services were incorporated properly. The application level allows for major performance boosts and minimal performance overhead.

4.3.2 Generation of Plaintext for Testing

We have created a testing harness that runs according to the configuration files that are passed to it. The harness also includes a program that generates the needed files for a given experiment. Below is how to generate the files using our distribution of Secdoop:
Generate your input sets and configuration files by calling the generate.py script in the secdoop directory:

```
./generator.py
```

If you would like, take a look at generator.py's usage by running:

```
./generator.py -h
```

Next, format the Hadoop namenode by running the following:

```
hdfs namenode -format
```

After running generator.py and formatting your namenode, you run the stage.sh script. This will put all of your generated input files into HDFS. This will create the necessary XML and text files for testing. After they run, the inputs and configs directories will be populated. (See more included in Appendix A)

4.3.3 Testing and Validation

Unit tests were written to ensure that the ciphers were implemented correctly and returned the expected results. In order to validate that Secdoop was performing the cryptography as expected, we used the Gutenberg example files that is often used for the "Wordcount" example for Hadoop. The input file for encryption was the Gutenberg ebook of Ulysses by James Joyce. The resulting file from the encryption application was then sent as the input to the decryption application. The output of the decryption was then compared to the original input. We used this method for the Single Machine applications as well as the Hadoop applications that are tested during our experiments. After the validation tests were returned with the expected results, we generated test files to be used as benchmark testing. Configurations were also generated in order to help with the automation of testing. Configuration files included the file size to be tested, the algorithm to be tested, and the location of the input files. The original input, encrypted data, and decrypted data reside on the cluster after the experiments have completed. All of the data is stored in part files while on the HDFS, but a simple script can restore these back to normal if the solution calls for a piece of intermediate data.

4.4 Implementation Details

4.4.1 Hadoop Application Behavior

When a Hadoop application starts up, the NameNode, or master node, locates which nodes already have the data replicated on the worker node's local storage. The input files are split up into many parts called “Input Splits” and then passed individually to the application’s mapper class. There are many records in each input split, and by default, each record consists of one line from the original input file. The mapper then sends one record at a time to a map function that is custom to the application. After initial unit tests were run on our proof of concept mapper class, we found that in the “LineRecordReader” class, the byte buffer that stores the records for each input split was not creating a new instance of the byte buffer for each new record. This did not clear the buffer and left unexpected data in the buffer if the next record was shorter than the previous record. The reason that Hadoop does not create the new byte buffer for each record is because of the overhead that it would cause. We solved this problem by using a “System.arraycopy()” to create the new byte buffer for each record in our mapper class and strip the excess of previous records.

4.4.2 Secdoop Application Details

In our experiments we employ the use of password based cryptography in order to ensure that the keys will be the same for every mapper class without using key management. By using password based cryptography we are able to use the similar variations of the passphrases for each of the three algorithms. In order to limit the differences between the three algorithms, we also used Cipher Block Chaining mode for all three standards.

The first application will consist of a driver and a mapper to implement encryption. The driver program will setup the configurations needed to run the mapper. The mapper class will run a static block of code that will set up a cipher stream for each input split. The map function will be called for each record in the input split and then encrypt it using one of the three algorithms specified above. The input key/value pairs will be of type LongWritable for the key, and of type Text for the values. The output key/value pairs will be of type LongWritable for the key, and BytesWritable for the values. We use the Hadoop api for LongWritable, Text, and BytesWritable. The map function will encrypt the input values and associate the output values with a key that will keep the sequence of the lines in the original file. The encryption application does not need a reducer function because we do not plan on combining any information. The output will be an intermediate file that will consist of the key value pairs. The output file format will be a sequence file, which can also be found in the Hadoop api, and this will store the data in an intermediate file on the HDFS.

The second application will consist of a driver and mapper for the decryption process. The driver will act the same as the previous by setting up the configuration and file types to be dealt with. The mapper will run a static block of code to set up the cryptographic keys and ciphers for decryption. The input file will be an intermediate sequence file input format that is a result from the first application. The input key/value pairs for the decrypt mapper will be of type LongWritable for the keys, and the values will be of type BytesWritable. The output key/value pairs will be of type LongWritable for the keys, and Text for the values. The map function will go through each line and decrypt using the ciphers set up in the static block. The values are still associated with a key that will keep the order of the original file, and thus a reducer function is not needed. The results will be plaintext that is the same as the original input file.

Chapter 5

Experiments

5.1 Experimental Design

This experiment is designed to demonstrate the difference in runtime between a single machine and a Hadoop cluster environment consisting of ten identical nodes. For the AES, DES, and DESede algorithms, we used the cipher-block chaining mode in order to keep a consistent mode for each algorithm and to limit the number of variables in the experiment. We used password based encryption in order to guarantee that the key would be the same during every trial without the need for using key management. Unit tests were written for each program used in the tests in order to verify the integrity of the results.

For the single machine trails, three algorithms and eight different files sizes were used. For each of the three algorithms a program was created to encrypt the data and another to decrypt the data, totaling six programs. The trials consists of running each program for every file size.

For the Hadoop cluster environment trials, three algorithms and the same eight file sizes were used. Eight total programs were created for the cluster environment, consisting of two drivers and six mappers. Two drivers were created, one to configure Hadoop for encryption and the other to configure Hadoop for the decryption. Each of the three algorithms required a mapper for encryption and another for decryption, totaling six mappers. There is a configuration file which is passed in at the command line and specifies which algorithm's mapper to use. When the file is encrypted, we use the key from the key value pair to keep the sequence of the file intact, thus eliminating the need for a reducer. For a trial, each mapper runs for every file size.

5.2 Experimental Setup

Single Machine Configuration: HP ProLiant ML110G6 brand computer with a 2.4Ghz, 4 core, Intel Xeon processor, 2GB of RAM, and 160GB HDD, running Ubuntu 10.04 (32 bit version). There is no cluster or network overhead. The machine is functioning as a local, stand-alone unit, and is considered to be a comodity machine due to it's common hardware and affordable price.

Cluster Configuration: Ten machines with hardware identical to the single machine using a Gigabit switch, comprise the ten node cluster used for experiments.

File Sizes: 1MB, 64MB, 512MB, 1GB, 2GB, 4GB, 8GB, 32GB.

Data Collected: All data collected for use in figures and graphs are an average of three experimental trials and are expressed in seconds. Due to minute differences in the encryption and decryption timing, only the encryption timings are presented in the figures and graphs.

5.3 Experimental Results

Figure 5.1 shows the average runtime of all three cryptographic algorithms running on a single machine. The runtime increases exponentially as the size of the file increases. AES has the fastest runtimes by far when compared to the other two algorithms. DESede unexpectedly outperformed DES on the 1MB file, but this was due to the limited number of trials run. The three algorithms behavior's are clearly demonstrated even with a limited number of trials run, and as the file sizes get larger, the difference becomes more and more apparent. Table 5.1 lists the data that was used for Figure 5.1. The choice of algorithms becomes a major performance decision as file sizes increase. For a 32GB file, AES completes encryption in almost one third of the time that DESede takes.

The three algorithms are much more competitive when running in the parallel environment on the cluster. Figure 5.2 compares the three algorithms on a ten node cluster, and shows that the parallel environment enhances the runtime significantly. The cluster overhead becomes a large performance hit when the size of the files are below 512MB. The three algorithms maintain the performance behaviors demonstrated in the single machine, but the difference between the performance is relatively minimal for all file sizes.

The figures show that the cluster is an ideal solution for large files. Figure 5.3 compares the performance of AES on the single machine and the ten node cluster. The single machine outperforms the cluster for the 1MB file and the 64MB file. The cluster significantly outperforms the single machine for any file larger than 512MB. The performance gain from the cluster increases as the file sizes increase. Table 5.2 lists the data points that were gathered from the cluster testing. When comparing the Table 5.1 and Table 5.2, the cluster completes the AES encryption for the 8GB file 3.69 times faster than the single machine, and the 32GB AES encryption completes 4.34 times faster than the single machine.

Figure 5.4 compares the DES algorithm on the single machine and the ten node cluster. The performance gains from the cluster is greater than the AES performance gains on the cluster. Similar to AES, the performance gain is not recognized until the 512MB file size is reached. The cluster completes the DES encryption for the 8GB file 6.35 times faster than the single machine, and the 32GB file completes 7.07 times faster than the single machine.

On the single machine, DESede was by far the slowest of the three algorithms. Hadoop enables DESede to be a competitor against the other algorithms. Figure 5.5 shows the dramatic performance gain, when using the cluster compared to the single machine, on large file sizes. The DESede completes encryption for the 8GB file 8.76 times faster than on the single machine, and the 32GB file completes 9.60 times faster than the single machine.

We noticed during testing that only half of the CPU was being utilized, so we investigated if there was possible performance tuning by adding more map tasks to be run simultaneously on the worker nodes. Tests were set up to compare the three algorithms with the maximum map tasks set to be 2, 3, and 4. These are simple configurations that can be added at the command line or in a configuration file. Table 5.3 demonstrates the effects of

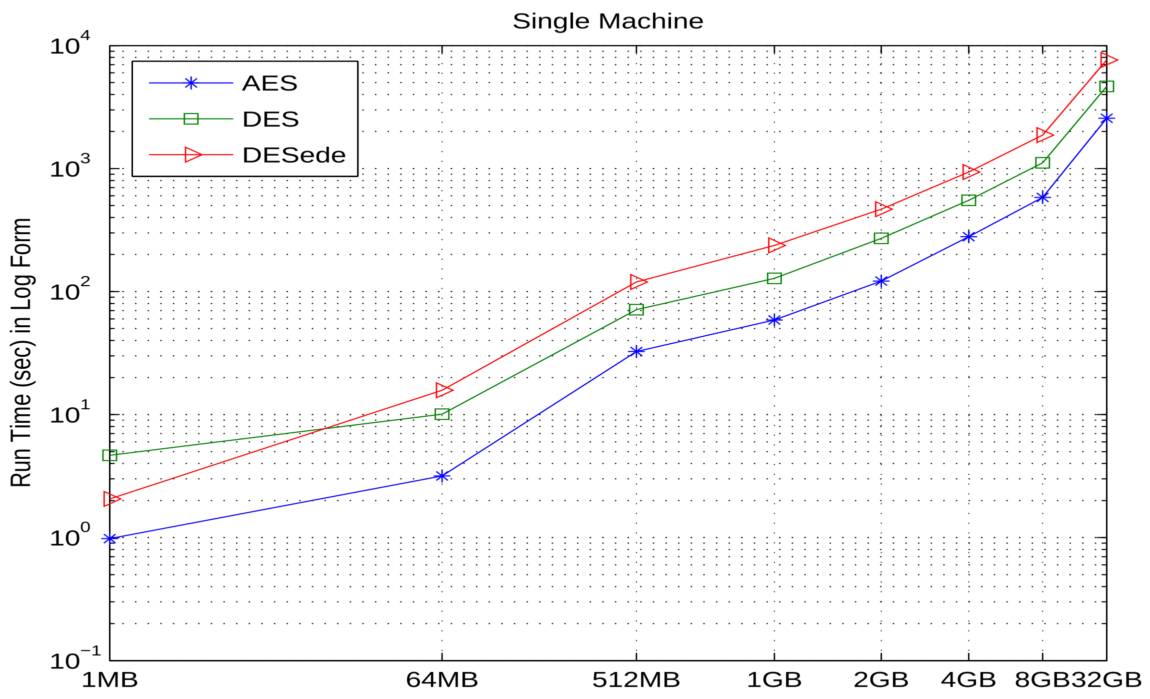

Figure 5.1: Single Machine Encrypt Test Runtimes with All Algorithms

the number of maps for the AES algorithm. The results are inconclusive and do not show much of any pattern except for the 32GB file. Table 5.4 shows the effects of the number of map tasks for the DES algorithm. The data collected for DES is also inconclusive and the only pattern is the 32GB file. Similarly Table 5.5 shows the results from the DESede tests for the number of map tasks effect on the algorithm. The gain for the 32GB file is encouraging in the fact that future tests should be done in order to see the effect on larger files than 32GB.

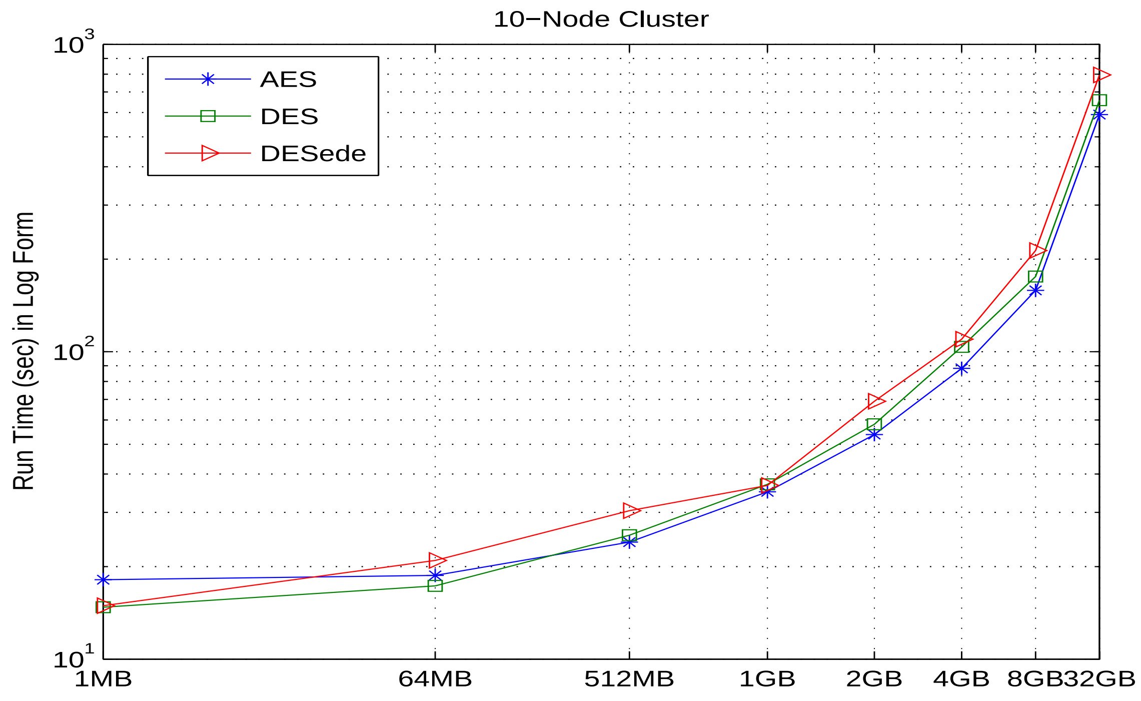

Figure 5.2: Ten Node Cluster Encrypt Test Runtimes with All Algorithms

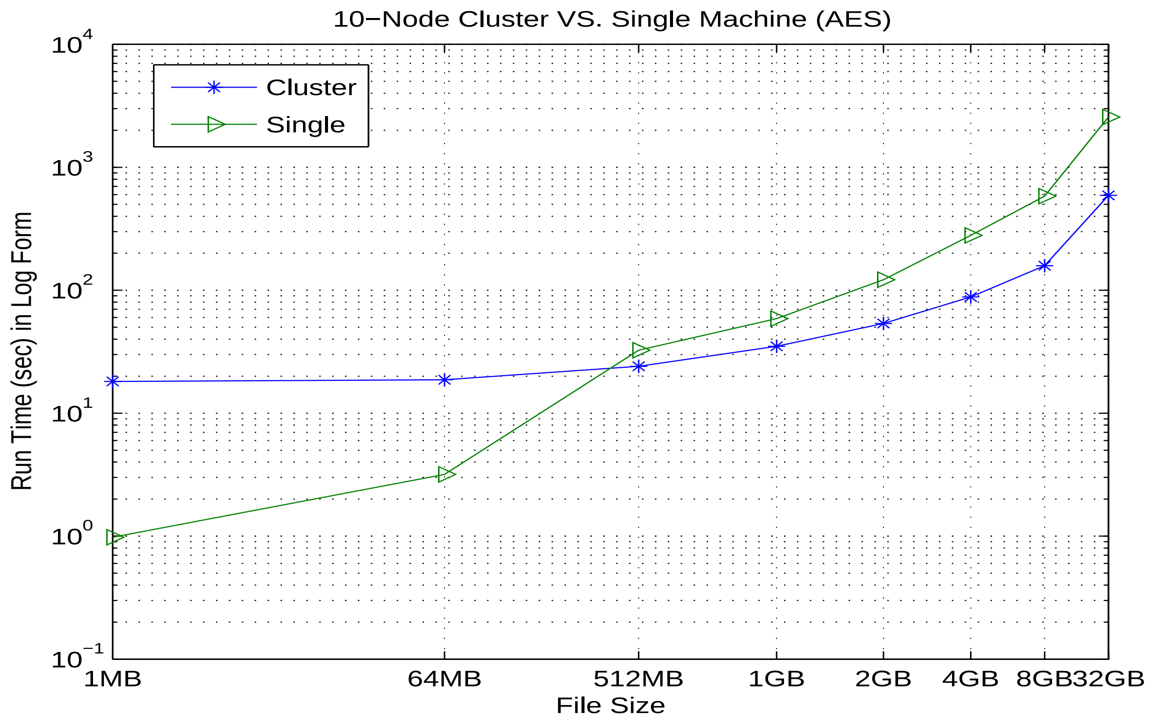

Figure 5.3: AES Encrypt Test Runtime Comparisons of the Cluster vs. Single Machine

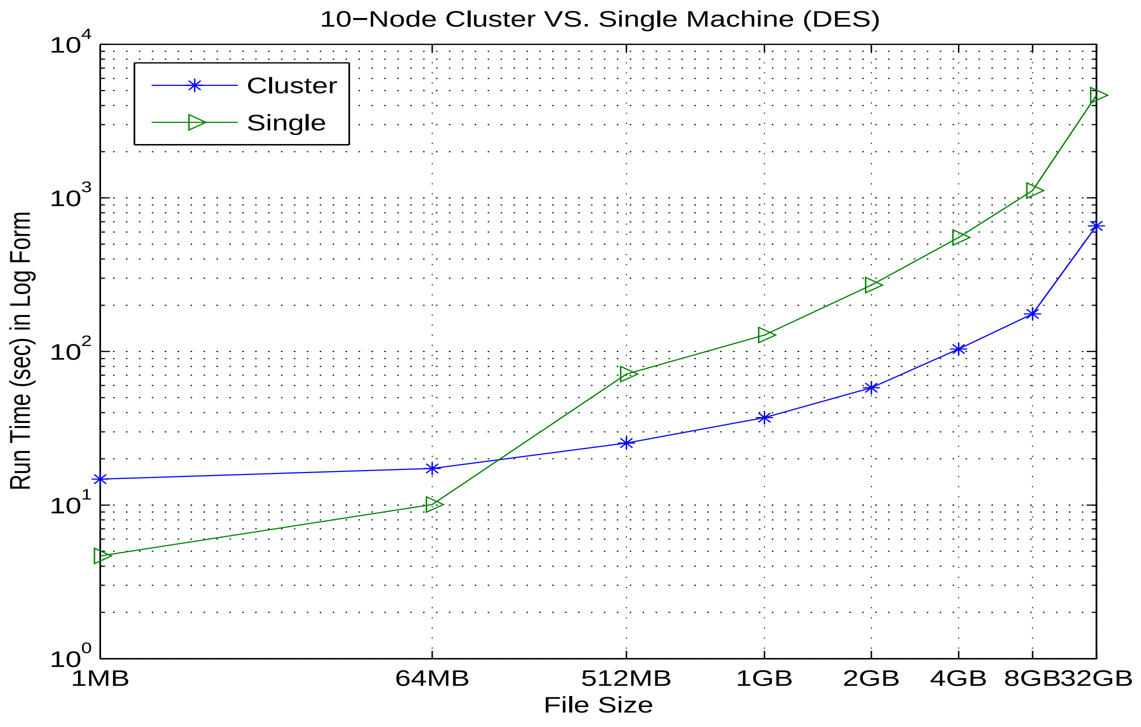

Figure 5.4: DES Encrypt Test Runtime Comparisons of the Cluster vs. Single Machine

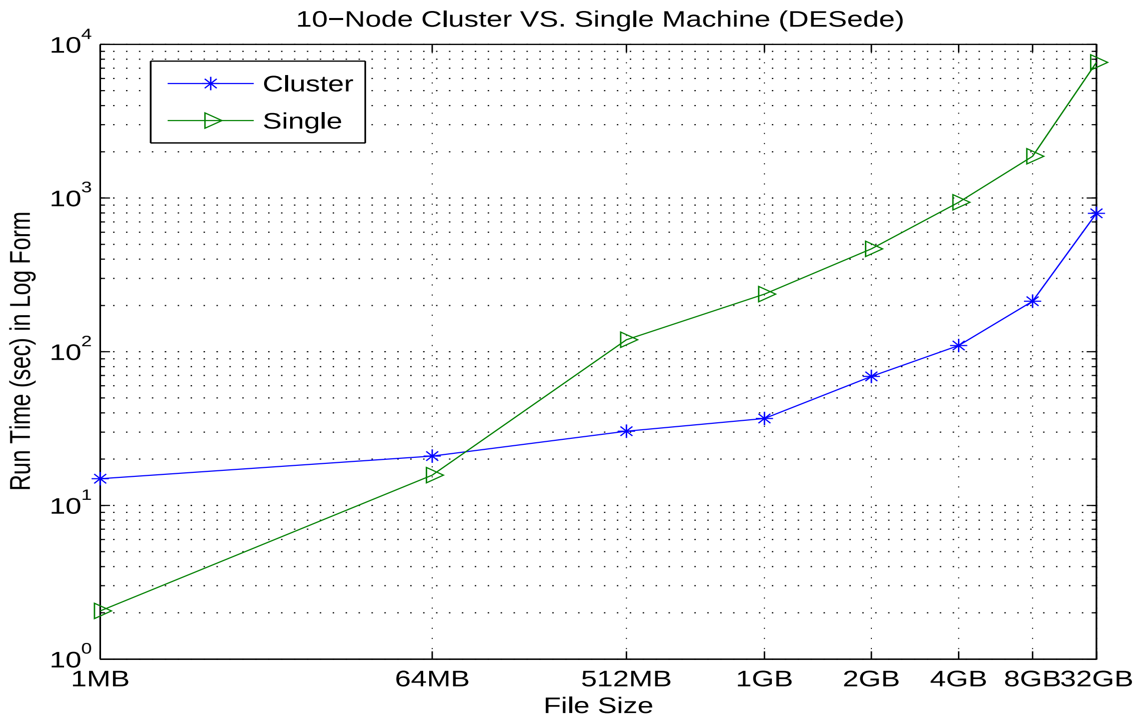

Figure 5.5: DESede Encrypt Test Runtime Comparisons of the Cluster vs. Single Machine

File Size	AES	DES	DESede
1MB	0.98	4.67	2.06
64MB	3.17	10.07	15.73
512MB	32.52	71.29	119.8
1GB	58.7	127.94	236.96
2GB	121.71	270.56	466.77
4GB	279.52	551.75	937.85
8GB	583.29	1115.04	1868.93
32GB	2566.9	4654.84	7637.29

Table 5.1: Average Encrypt Runtimes on a Single Machine in Seconds

File Size	AES	DES	DESede
1MB	18.13	14.75	14.92
64MB	18.71	17.31	20.95
512MB	24.04	25.31	30.41
1GB	35.04	37.04	36.74
2GB	53.74	58.08	69.01
4GB	88.23	103.77	109.8
8GB	158.27	175.59	213.37
32GB	591.18	657.99	795.52

Table 5.2: Average Encrypt Runtimes on a Ten Node Cluster in Seconds

File Size	2 Maps	3 Maps	4 Maps
1MB	18.13	16.3	15.3
64MB	18.71	17.31	17.46
512MB	24.04	25.72	23.98
1GB	35.04	34.32	35.2
2GB	53.74	51.55	52.1
4GB	88.23	79.65	93.75
8GB	158.27	144.19	149.54
32GB	591.18	564.98	534.24

Table 5.3: AES Encrypt Runtime Comparisons between the Maximum Number of Map Tasks

File Size	2 Maps	3 Maps	4 Maps
1MB	14.75	14.71	14.68
64MB	17.31	17.97	17.27
512MB	25.31	26.97	25.39
1GB	37.04	32.95	32.75
2GB	58.08	45.91	54.49
4GB	103.77	92.25	86.78
8GB	175.59	175.74	154.6
32GB	657.99	597.5	583.72

Table 5.4: DES Encrypt Runtime Comparisons between the Maximum Number of Map Tasks

File Size	2 Maps	3 Maps	4 Maps
1MB	14.92	14.76	14.86
64MB	20.95	20.28	20.28
512MB	30.41	30.76	29.85
1GB	36.74	38.38	40.65
2GB	69.01	60.55	60.44
4GB	109.8	102.2	98.75
8GB	213.37	194.59	177.97
32GB	795.52	707.14	679.53

Table 5.5: DESede Encrypt Runtime Comparisons between the Maximum Number of Map Tasks

Chapter 6

Conclusion

The results from the experiments show exponential improvement in the processing speed as the files increase in size. The AES algorithm outperformed the DES and DESede algorithm as expected, but the results on the cluster were much closer than the results of the single machine. The single machine outperformed the cluster when the file sizes were less than 512MB, but there was an exponential gain when using the cluster as the file sizes increased above 512MB. This solution is optimal for companies that have large amounts of sensitive data that need to be distributed. This solution can be implemented where companies can own a commodity cluster at a low cost, and prepare large sets of data for further distribution. Our model, Secdoop, is an extremely useful model when industries are concerned about the security of the cloud. If the data is already encrypted by our solution, then industries can store data and not worry about where the data resides in the cloud. Secdoop provides an option to process securely over an in-house cluster that is scalable and cost effective.

There is room for performance tuning by manipulating the configuration of Hadoop on a cluster. We are interested in the comparison of runtimes when file sizes are larger than 32GB and we increase the Maximum Number of Map Tasks to be closer to the number of cores per CPU in each node. The results that we gathered using the two, three, and four Maximum Number of Map Tasks was not enough to conclude any patterns. We also plan on comparing the effect of the number of nodes in a cluster versus the performance gain over a single machine.

Chapter 7
Future Work

This project has been documented in order to serve as an example to future students in the field of computer science. We have created project descriptions and lectures that will aid in the education process. The project descriptions and lesson plans cover topics such as developing tests for applications, developing applications in a distributed environment, and how to implement security measures in Java.

This project is the framework for a security suite that has started with cryptography and will focus on topics such as: key management, authorization, and authentication. Another possibility is to create a library of configurations that holds the settings for the different algorithms that will be used by this application. With a library of preconfigured settings, companies could download this suite and run it on their own secure in-house cluster without the need for customization. If a company desired a customized configuration, then they could use our library of solutions as an example.

There is also an ethical negative side to the use of MapReduce if the mappers are written to attack a set of data. This leads to the need for stronger encryption or larger keys. If distribution of processing can break keys faster than before, then all of the data that is currently thought of as “safe” will need stronger protection. This is alarming when clusters are becoming more and more affordable.

Bibliography

[1] Dean, J., Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM - 50th anniversary issue: 1958 - 2008, volume 51, issue 1 (January 2008), pg 107-113*

[2] Shvachko, K., Kuang, H., Radia, S., Chansler, R., "The Hadoop Distributed File System," *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), p.1-10, May 03-07, 2010*

[3] Krishna, M., Kannan, B., Ramani, A., Sathish, S., "Implementation and Performance Evaluation of a Hybrid Distributed System for Storing and Processing Images from the Web," *Cloud Computing Technology and Science, IEEE International Conference on, pp. 762-767, 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010*

[4] Jia, B., Wlodarczyk, T., Rong, C., "Performance Considerations of Data Acquisition in Hadoop System," *Cloud Computing Technology and Science, IEEE International Conference on, pp. 545-549, 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010*

[5] White, T, Hadoop: The Definitive Guide. *O'Reilly Media, 2009.*

[6] Park, S., Lee, J., Chung, T., "Cluster-Based Trust Model against Attacks in Ad-Hoc Networks," *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on , vol.1, no., pp.526-532, 11-13 Nov. 2008*

[7] Mishra, S., Kushwaha, D.S., Misra, A.K., "A Cooperative Trust Management Framework for Load Balancing in Cluster Based Distributed Systems," *Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on , vol., no., pp.121-125, 12-13 March 2010*

[8] Li, X., Jing, Z., "A Trust Cluster Based Key Management Protocol for Ad hoc Networks," *Anti-counterfeiting, Security, Identification, 2007 IEEE International Workshop on , vol., no., pp.371-376, 16-18 April 2007*

[9] O'Connell, M., Nixon, M., "JFS: A Secure Distributed File System for Network Computers," *EUROMICRO Conference, 1999. Proceedings. 25th , vol.2, no., pp.450-457 vol.2, 1999*

[10] Sandberg, R., "The Sun Network File System: Design, Implementation and Experience," *Tech. Report, Mountain View CA: Sun Microsystems, 1987*

[11] LeBlanc, T., Subhlok, J., Gabriel, E., “A High-Level Interpreted MPI Library for Parallel Computing in Volunteer Environments,” *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID ’10). IEEE Computer Society, Washington, DC, USA, 673-678.*

[12] Aleksander, M., Litawa, G., Karpinskyi, V., “Distributed computing system which solve an elliptic curve discrete logarithm problem,” *CAD Systems in Microelectronics, 2009. CADSM 2009. 10th International Conference - The Experience of Designing and Application of , vol., no., pp.378-380, 24-28 Feb. 2009*

[13] Gomes, A.M., Kakugawa, F.R., de Paula Bianchini, C., Massetto, F.I., “A thread-safe communication mechanism for message-passing interface based on MPI Standard,” *Pervasive Computing (JCPC), 2009 Joint Conferences on , vol., no., pp.173-178, 3-5 Dec. 2009*

[14] Prabhakar, R., Patrick, C., Kandemir, M., “MPISec I/O: Providing Data Confidentiality in MPI-I/O,” *Cluster Computing and the Grid, 2009. CCGRID ’09. 9th IEEE/ACM International Symposium on , vol., no., pp.388-395, 18-21 May 2009*

[15] Leo, S., Santoni, F., Zanetti, G., “Biodoop: Bioinformatics on Hadoop,” *ICPPW, pp.415-422, 2009 International Conference on Parallel Processing Workshops, 2009*

[16] Wei, W., Du, J., Yu, T., Gu, X., “SecureMR: A Service Integrity Assurance Framework for MapReduce,” *Computer Security Applications Conference, 2009. ACSAC ’09. Annual , vol., no., pp.73-82, 7-11 Dec. 2009*

[17] National Institute of Standards and Technology, Federal Information Processing Standards Publication 197, “Advanced Encryption Standard,” *November 26 2001*

[18] National Bureau of Standards, Federal Information Processing Standards Publication 46-3, “Data Encryption Standard,” *National Bureau of Standards, US Dept. of Commerce, Jan. 1977*

[19] Hou, F., Gu, D., Xiao, N., Tang, Y., “Secure Remote Storage through Authenticated Encryption,” *Proceedings of the 2008 International Conference on Networking, Architecture, and Storage (NAS ’08). IEEE Computer Society, Washington, DC, USA, pg 3-9*

[20] Gu, Y., Grossman, R., “Exploring Data Parallelism and Locality in Wide Area Networks,” *Workshop on Many-task Computing on Grids and Supercomputers (MTAGS), co-located with SC08, Austin, TX. Nov. 2008*

Appendices

Appendix A

Secdoop Subversion Information

We are using Subversion for version control. To checkout the repository, it is suggested that you add the following environment variable to your .bashrc:

```
export SECDOOP_SVN=http://<username>@svn.binary-snobbery.com/secdoop
```

Where <username> is your Subversion username.
-- NOTE: This may change in the future to allow for anonymous SVN access.

To checkout the Subversion repository, in your HOME directory run:

```
svn co $SECDOOP_SVN/trunk trunk
```

NOTE: Checking out in another location will require you edit the following environment variables to point to the correct location.

You must add the following lines to your .bashrc file:

```
export HADOOP_DIR=$HOME/trunk/hadoop
export HADOOP_COMMON_HOME=$HADOOP_DIR
PATH=$PATH:$HADOOP_DIR/bin
```

HADOOP_COMMON_HOME is used by the Hadoop scripts.
HADOOP_DIR is used by the Secdoop scripts.

To build Secdoop, you can simply run 'ant dist-all' in the trunk/secdoop folder. This will compile all of the necessary classes needed to test both Secdoop and the standalone JCE tests.

In order to test anything against the Java Cryptographic Extensions (JCE), you need to install the JAR files from the jce_policy-6.zip file in trunk/. Unzip the file and copy it into your JRE's lib/security director. For example, on Ubuntu 10.* you can do the following:

```
unzip jce_policy-6.zip
```

```
sudo cp jce/*.jar /usr/lib/jvm/java-6-sun/jre/lib/security
```

First thing's first, actually configure hadoop. You'll find some example configuration files in hadoop/conf. Copy those files without "example" in the name, and you'll have the files you need for configuration. Change the values to whatever is appropriate for your cluster.

The testing harness is configured through a tests configuration file, by default named "tests.conf." In tests.conf you can specify the algorithms you wish to test and the size of the input files. Currently supported sizes and algorithms are listed in tests-example.conf.

Then, you need to generate your input sets and configuration files by calling the generate.py script in the secdMishra, S.; Kushwaha, D.S.; Misra, A.K.; ,

```
./generator.py
```

If you would like, take a look at generator.py's usage by running:

```
./generator.py -h
```

Next, format the Hadoop namenode by running the following:

```
hdfs namenode -format
```

After running generator.py and formatting your namenode, you run the stage.sh script. This will put all of your generated input files into HDFS.

This will create the necessary XML and text files for testing. After they run, the inputs and configs directories will be populated.

Next, start up Hadoop:

```
hadoop-startup.sh
```

NOTE: In our testing environment, we have a shared home directory via NFS. So, each of the Hadoop installations is actually shared across the cluster. This can be problematic, because the hadoop log directory is on NFS. The overhead from NFS writes was actually enough to notice a roughly 10% performance decrease. Also, the userlogs directory had to by symbolically linked to a common local (non-nfs) directory. If your cluster doesn't share a Hadoop installation, this shouldn't be an issue.

Then you're ready to run tests. From the secdoop directory:

```
./test.sh
```

For information about test.sh command-line options, simply run:

```
./test.sh -h
```

After testing, shutdown the Hadoop cluster:

```
hadoop-shutdown.sh
```

If at any point you need to refresh your Hadoop cluster (i.e. delete all of your temporary mapreduce data, remove all of your HDFS local file stores on cluster nodes, and re-format your namenode), you can use hadoop-clear.sh found in hadoop/bin. This script should be in your PATH if you made the changes to your .bashrc, mentioned above, correctly.

Appendix B
Example Mapper Class

```
/***************************************************************************/
/* EncryptMapper.java by James Majors                                     */
/* This is the example Mapper function for the AES Encryption application. */
/* The input will be split up and equally distributed across the available */
/* nodes in the cluster.  The encryption process will be done for each     */
/* input split and paired with a key value that will help maintain the     */
/* sequence of the file.  The output will be sent to an intermediate file  */
/* that will contain the key/value pairs.                                  */
/***************************************************************************/

public class EncryptMapper extends MapReduceBase
  implements Mapper<Input Key1, Input Value1, Output Key2, Output Value2> {

  private static Cipher encryptCipher;
  static {
//GENERATE KEY FOR THE GIVEN ALGORITHM
//INITIALIZE CIPHER
   }


  public void map(Key1, Value1,
      OutputCollector<Key2, Value2> output, Reporter reporter)
      throws IOException {
//Replace the value from the input splits with the System.arraycopy()
//Encrypt the input value
//Output the results to an intermediate file (<Key2, Value2>)
  }
}
```

Appendix C
Configuration File Example

This is an example of the configuration for the AES 1MB test:

```xml
<?xml version="1.0"?>
<configuration>
    <property>
        <name>inputFile</name>
        <value>inputs/1MB.txt</value>
    </property>
    <property>
        <name>encryptedFile</name>
        <value>AES-1MB-enc</value>
    </property>
    <property>
        <name>decryptedFile</name>
        <value>AES-1MB-dec</value>
    </property>
    <property>
        <name>algorithm</name>
        <value>AES</value>
    </property>
</configuration>
```