TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

PROTECTING BIG DATA STORAGE WITH TRUSTED COMPUTING:

DESIGNING A TRUSTED INFRASTRUCTURE AND SOFTWARE SOLUTION

by

Jason Cohen

A Dissertation

Presented to the faculty of

Towson University

in partial fulfillment

of the requirements for the degree

Doctor of Science, Applied Information Technology

Department of Computer Science

Towson University
Towson, Maryland 21252

May, 2014

TOWSON UNIVERSITY

COLLEGE OF GRADUATE STUDIES AND RESEARCH

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation thesis prepared by Jason C. Cohen entitled "PROTECTING BIG DATA STORAGE WITH TRUSTED COMPUTING: DESIGNING A TRUSTED INFRASTRUCTURE AND SOFTWARE SOLUTION" has been approved by this committee as satisfactory completion of the requirement for the degree of Doctor of Science in Information Technology.

Dr. Subrata Acharya Chair, Dissertation Committee	Subrata Acharya Signature	05/14/14 Date
Dr. Josh Dehlinger Committee Member	 Signature	5-19-14 Date
Dr. Michael P. McGuire Committee Member	 Signature	5-20-2014 Date
Mr. Rick Supplee External Committee Member	Richard Supplee Signature	5/19/14 Date
Mr. Michael Moore External Committee Member	Michael R. Moore Signature	5/19/14 Date
Dr. Janet Y. DeLany Dean, College of Graduate Studies and Research	Janet V DeLany Signature	5/19/14 Date

Abstract

Protecting Big Data Storage with Trusted Computing: Designing a Trusted Infrastructure and Software Solution

Jason C. Cohen

Apache Hadoop has the potential to offer powerful and cost effective solutions to big data analytics; however, sensitive data stored within a Hadoop Distributed File System (HDFS) infrastructure has equal potential to be an attractive target for exfiltration, corruption, unauthorized access, and modification. Pairing Apache Hadoop distributed file storage with hardware-based Trusted Computing mechanisms, based on TCG standards, has the potential to alleviate risk of data compromise. With the growing use of Hadoop to tackle big data analytics involving sensitive data, an HDFS cluster could be a target for data exfiltration, corruption, or modification. By implementing open, standards based Trusted Computing technology at the infrastructure and application levels; a novel and robust security posture and protection is presented. A discussion of the motivation for research on this topic, a threat model and evaluation of a targeted Advanced Persistent Threat against HDFS is presented, and a set of common security concerns within HDFS is addressed through infrastructure and software involving

integrity validation and data-at-rest encryption. To accomplish these goals, technology from the Trusted Computing Group, such as the pervasively available Trusted Platform Module is used. In addition, a discussion of design considerations in building an encryption framework for Hadoop in a trustworthy manner is presented along with a description of performance and security results of experiments creating an encryption scheme for Hadoop utilizing hardware key protections and Intel AES-NI (Advanced Encryption Standard New Instructions) for encryption acceleration. This work includes an evaluation of the recently implemented crypto framework for Hadoop and independent test of the performance claims of AES-NI regarding mitigating encryption performance overhead.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

INTRODUCTION AND MOTIVATION

"Big data" has become a hot topic in the enterprise space within recent years, perhaps even rivaling the popularity of "cloud" in the buzzword count within popular trade literature. But, what does "big data" mean, and what are the implications for an organization's security posture when using popular platforms to store and process this data? "Big data" is a singular term used to describe an array of possibilities, with different meanings applying in different organizations. In the general case, it means being able to store and reason about large quantities of both structured and unstructured data. For instance, in the world of a search giant like Google, this means generating near-instant responses to search queries from an enormous amount of raw data drawn from an enormous quantity of web pages. There are a number of solutions in the market for storing and operating on big data, however, Apache Hadoop has brought this capability to the masses through non-proprietary, open-source software that can scale to enormous proportions. For these reasons, Apache Hadoop was selected as the base platform for this research in bid data security.

A 2012 *Harvard Business Review* survey queried executives at Fortune 1000 companies regarding their plans and expectations for using big data in their enterprises. Of the organizations surveyed, 85 percent reported that they had big data initiatives planned or in progress, 70 percent reported that these initiatives were enterprise-driven, 85 percent of the initiatives were sponsored by a C-level executive or the head of a line of business, and 75 percent expected an impact across multiple lines of business [1]. This survey demonstrates the strong and growing desire for organizations to make better use of their data and tap into the unknown treasures therein through big data analytics. The

open-source Apache Hadoop project has become a prime contender in the big data solution space, with numerous vendors stepping up with offerings that are based on it. Although there are a number of applications that can sit on top of Hadoop, the core services that it provides are a distributed storage framework called Hadoop Distributed File System (HDFS) and a distributed application framework called MapReduce. The core goal is improved data processing speed on large datasets. To achieve this goal, the system splits large data sets into smaller pieces, distributes them to as many storage/processing nodes as possible, and processes the data so that processing and data are tightly coupled, with the resulting output being aggregated. By distributing data and processing to a network of commodity servers, Hadoop allows large, computationally difficult data to be stored and analyzed in an efficient way. For certain application domains, the Hadoop framework can provide a more efficient and cost-effective approach when compared to other available commercial solutions, particularly considering that it has the ability to outsource and distribute processing and storage resources. MapReduce, a programming concept that takes a large problem, breaks it down into smaller problems, and distributes those small units and associated data, is the core concept of Hadoop. Taking the processing to the data by executing code where the data actually resides eliminates data throughput restrictions for remotely-located or centrally-located data. As such, it provides an efficient and elegant way to deal with operations on large datasets. The framework was also designed to provide data redundancy without any special hardware, instead relying on basic servers rather than large-scale systems, virtualization, etc. [2]. Processing speed and data redundancy were the core concepts behind Hadoop's development, with security concerns being a secondary concern and primarily an

afterthought. The addition of a basic security framework in 2009 did not seriously address all security concerns, as the designers made certain assumptions about how Hadoop would be used and how it would be isolated from the network at large [3]. Confirming the perception of a security deficiency, a recent survey of Hadoop users indicated that security was the primary area in which they desire improvement [4].

Given the wide range of applications for big data, it is not hard to imagine that sensitive data could end up in an organization's HDFS infrastructure, making it a ripe target for security exploitation. Depending on the organization's business, this data could represent a treasure trove of information in which a potential adversary could be acutely interested. Of course, typically, an organization will not store sensitive engineering files, passwords, trade secrets, etc., in an HDFS, as it is not a general-purpose file system. That is not to say, however, that information related to an organization's sensitive activities could not find its way there—and if an organization can gain financial benefit from the data stored within an HDFS, someone else may be able to do so as well. To take an obvious example, a defense organization could use Hadoop to store and process raw intelligence information. Being able to access or alter this information, or simply being able to determine the type of information being stored, could be of extreme benefit to an adversarial nation state. In the context of healthcare, Hadoop could be used to aid in fraud detection within claims and conduct research analytics on medical records, as well as to execute a number of other activities involving personally identifiable information; a breach in this information would be a violation of legal regulations, and damaging to the reputation of the data owner. Organizations have an obligation to protect sensitive data beyond Hadoop's default mechanisms. Another interesting capability of Hadoop that is

being explored within some organizations is the use of its storage and analytic capabilities in network defense. Typically this means gathering the log files of all network components, security appliances, access logs, host logs, etc., into Hadoop and conducting event correlation and anomaly detection. This capability has the potential to aid in the detection and prevention of threats; on the other hand, it also presents another potential place that a threat could target without detection.

With the prevalence of cloud outsourcing, external hackers, and insider threats, as well as legislation requiring organizations to make efforts to protect certain data, a need has arisen for a better security framework to protect data within HDFS. Given a distributed Hadoop cloud spanning datacenters, remote partners, or potentially untrustworthy sites, how can one be sure that the confidentiality and integrity of the data, as well as the integrity of the Hadoop binaries and underlying operating systems, is preserved? In other words, how can one "trust" the platform to which one is submitting sensitive data? How can one be sure that this data will not be altered when the system is offline? Hadoop assumes that any network involved in HDFS block transfer is secure and not publicly accessible for sniffing [3]. Considering these limitations, and the potential of Hadoop to address problem sets involving sensitive data including protected information such as health informatics, defense, and personal data, the need to address the security of a Hadoop IT infrastructure is acute. This problem escalates when the Hadoop environment is distributed (i.e. among remote datacenters, outsourced, etc.). An example may be a hospital system that shares a data processing environment with other centers, or a group of universities in which each manages part of a Hadoop cloud within its respective boundaries. Although Hadoop makes an effort to ensure that data is safe from

corruption and stored in a redundant fashion, like all software, it is only as trustworthy as the systems and architecture on which it executes, and the people who have access to it (both officially and unofficially). This means that a Hadoop node compromised by a malicious outsider or insider threat could alter data by circumventing any security checks within the software. This vulnerability exists because HDFS is a virtual file system, and actual data chunks are distributed across real file systems in an unencrypted format. Someone with access to data from a Hadoop name node that contains metadata about files, and to Hadoop tools and/or source code, could hunt down data in the cluster and alter it. Additionally, one could replace the Hadoop Java packages with new packages that relay, alter, or otherwise corrupt data. While a node is offline, these attacks could be conducted by a malicious insider, who could then restore the system without leaving any indication of the event. Furthermore, on a distributed network, an attacker may be able to monitor traffic and conduct man-in-the-middle or replay attacks on the data between client and data nodes. A hacker gaining root access could also read unencrypted files from compromised nodes. The mechanisms provided by commercial Trusted Computing technologies can go a long way towards mitigating these possibilities by creating a measurable, verifiable operating environment, aiding in the detection of compromise and enabling encryption with strong key protections.

This research explores the protection of big data storage through the creation of a trusted Apache storage platform through the use of Trusted Computing (TC) concepts at the infrastructure and software levels [5] [6]. When the idea was first conceived of creating an IT architecture for Hadoop and developing software changes that would increase its trustworthiness, basics like on-wire encryption and data-at-rest encryption

were not present in the Hadoop common base (although some vendors were providing bolt-on solutions). As of late 2012–2013, a wider recognition of Hadoop's security limitations has become apparent, as SASL-based (Simple Authentication and Security Layer) data-in-transit encryption has become available, and most recently, a data-at-rest cryptographic framework has been proposed and implemented as part of Intel's recent Hadoop distribution offering [7]. In addition, organizations are beginning to recognize that traditional security mechanisms are not sufficient to deal with so-called "Advanced Persistent Threats" (APTs), and they wish to provide greater protection against these and other threats, such as malicious insiders [8]. This research aims to add additional protection mechanisms to a Hadoop IT infrastructure and software stack to help combat these threats, and also seeks to determine the trade-offs in terms of security value versus lost performance. An additional contribution of this research is demonstrating how one can take off the shelf Trusted Computing components and use it in an application specific environment, as well as how to integrate key protections and integrity management into an application. The general method could be applied to a number of other applications, including noSQL style big data solutions like MongoDB.

1.1 Organization of Dissertation

In order to present the reader with the technical background information needed to understand this research, the following Chapter will present an overview of the key concepts of Apache Hadoop and the Trusted Computing Group technologies. Chapter two also includes a synopsis of the related research in the area of applying trusted computing to Apache Hadoop. To better understand the specific threats and vulnerabilities present in the Apache Hadoop storage architecture, Chapter three presents

a threat model, static code analysis results, and an example of an Advanced Persistent Threat targeting Hadoop. Chapter four presents the technical design details of an experimental data protection and platform integrity framework using a trusted computing stack. The approach includes an IT infrastructure element utilizing trusted computing concepts and an encryption engine with key protections rooted in trusted computing. Security improvements are also presented. Chapter five covers the preliminary performance data of the encryption implementation with hardware rooted key protections. The results can also serve as independent verification of the claims made by Intel regarding their crypto design using the AES-NI processor acceleration. Chapter six addresses alternative Hadoop encryption design concepts that account for performance limitations found during performance testing. In addition, the latest proposed crypto framework for Hadoop, and possibilities for integrating the concepts of hardware key management into it are discussed. Besides the direct applicability of the results to the Hadoop application domain, the presented design considerations, which use Trusted Computing Group concepts in infrastructure validation and purposeful software design, can easily be applied to other applications in similar fashions.

To summarize, the specific research contributions here will include:

- A motivation for Trusted Computing integration
- An example of Trusted Computing-enabled IT architecture for HDFS
- A discussion of possible software integrations of Trusted Computing features
- A threat model for Hadoop
- A description of an Advanced Persistent Threat (APT) against Apache Hadoop
- Static Code analysis results

- A description and implementation of a trusted storage layer for Hadoop HDFS through modifications of the Hadoop HDFS source code
- Performance implications and analysis of the impact of AES-IN
- Security testing results
- Concepts for alternative designs
- A discussion of real-world challenges.
- Showing how Trusted Computing concepts can be applied to an application specific IT infrastructure and problem

Overview of Concepts and Literature Review

My interest in researching practical application for Trusted Computing began in September of 2012 when I had the opportunity to attend the NSA Trusted Computing Conference and Exposition. Frankly, even after working in and studying the IT security field for over a decade, I really did not know much about the topic or the implications of the technology prior to this event. I was surprised at the level of attendance and general excitement around the topic. Granted, most attendees were likely involved with work with various US defense departments, but still, it seemed that this relatively simple concept of an embedded device and protocol stack that enables secure key storage, integrity management, and attestation held promise for a number of practical applications. Indeed, there were a number of vendors offering solutions and presentations involving practical applications such as trusted multi-tenant infrastructure, self-encrypting drives, geofencing, and secure end-user operating systems, among others. I began to wonder about other unique and interesting ways in which this technology could be applied that have not yet been examined. When I returned from this experience, I began to research the technology and enabling software, and worked on a small, proof-of-concept application utilizing a Trusted Platform Module (TPM). Thinking about other pragmatic applications for this technology, I considered the recent "hot topic" of big data, and the security implications for organizations looking to employee this technology to tease unknown insights out of their data. It quickly became apparent to me that there were several key security concepts lacking in one of the most popular big data frameworks, Apache Hadoop. In the nearly two years since I began looking at this concept, strides have been made within the community to address some of the more glaring omissions—

namely that of data-in-transit encryption and data-at-rest encryption. With that in mind, and with exposure to the topic of Advanced Persistent Threats, I decided that Hadoop, and its storage platform in particular, would be an interesting case study for the application of somewhat boiler-plate Trusted Computing components, as well as for a novel encryption mechanism that utilizes Trusted Computing to protect keys. I believe these concepts would ultimately make it very difficult to steal data in any way outside of direct manipulation of the Hadoop software, and add a compelling layer in a robust security architecture. I was not particularly surprised to find that at the time research was initiated on this topic (Spring 2012), there were no references found in scholarly journals relating specifically to implementing Trusted Computing as part of a Hadoop IT architecture and as part of a software design. Interestingly, a few works have now been published in parallel to my efforts, indicating a wider recognition of the potential of Trusted Computing to help with the real-world problem of deploying trusted big data services.

As mentioned, the key components of this research are the Apache Hadoop Distributed File System and Trusted Computing. The following literature review is intended to provide an overview of the key concepts of these components, and enough technical detail to aid in the understanding of the integration work that makes up this research. In addition, the recent works published in parallel to my own will be listed and briefly described.

2.1 Overview of Hadoop and Trusted Computing Concepts

The following will serve as an overview of the Hadoop distributed file system (HDFS) and Trusted Computing tools that use Trusted Computing Group standards,

which will be levied to create a more secure and trustworthy architecture foundation for executing a Hadoop cluster and integrating trusted storage into HDFS.

Apache Hadoop was originally created by Doug Cutting, with the intention of creating an open-source implementation of the concepts defined by Google in their papers on MapReduce. The goal was to create a highly scalable (to thousands of nodes) distributed computing environment that could store unstructured data and perform analytical operations on that data where it is stored. Apache Hadoop is composed of two distinct items: the Hadoop distribute file system, and the application processing framework MapReduce. Note that HDFS is not the only way to store files for use with the Hadoop processing engine, as it supports a number of different file systems [9]. In addition, Hadoop supports software that runs over HDFS to provide additional functionality. Apache HBASE is an example of this, providing real-time read/write access to "big tables" stored within Hadoop [9]. The MapReduce engine is not a focus of the security model, and is an area for future examination of the applicability of Trusted Computing components at a software level.

The terms "Trusted Computing" and "trust" in terms of computing mean different things to different people. The current context refers specifically to the technologies developed and promoted by the Trusted Computing Group (TCG). These technologies are designed to create systems in which the state of the software components can be determined if they are in a known, trusted state. This is accomplished through hardware components, encryption, and strong key protection. Trusted Computing technology can be leveraged for a number of purposes, but the focus of this research is to levy its ability to enforce known, secure, measurable base IT platforms for executing Hadoop nodes and

for strong protections of keys used directly within Hadoop code to create a trusted storage system with enhanced confidentiality and integrity.

2.1.1 The Hadoop File System Overview

Securing the data element of a Hadoop infrastructure with hardware-rooted trust is the core idea of this research and is accomplished through establishment of an experimental IT architecture using Trusted Computing and developing trusted storage within the HDFS code. The following synopsis is based on the HDFS description published by the Apache Hadoop project, which provides a high-level overview of how Hadoop stores and accesses data [2].

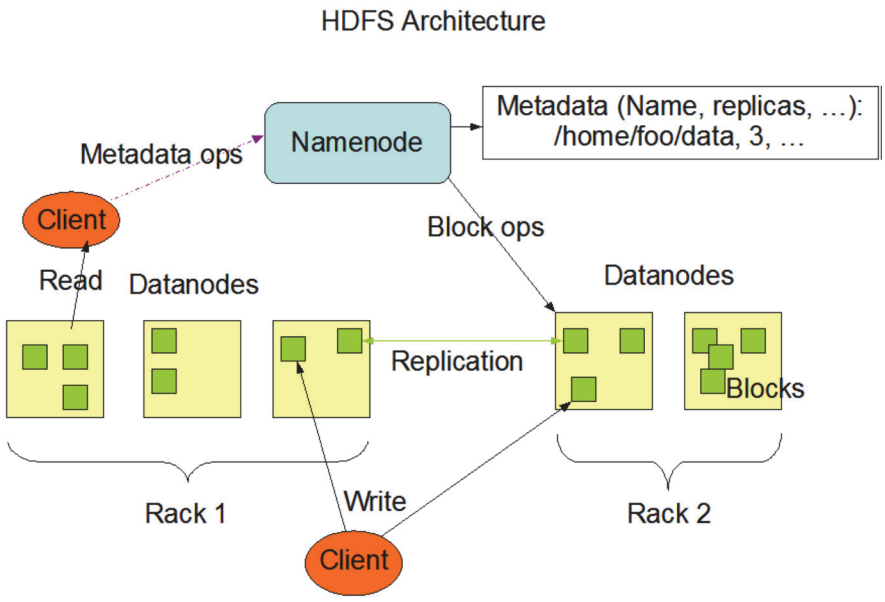

Figure 1. HDFS Architecture [2]

The Hadoop Distributed File System is a virtual files system, written in Java, which distributes chunks of files across Datanodes. The chunks, called blocks, are stored on the physical file system of each node in a configurable location. HDFS can be

configured to keep redundant copies of each block, to avoid loss of data even in the event of a node, or multiple node, failure. Hadoop has provisions for being "rack-aware," and can be configured to ensure that data is spread among multiple racks (and potentially multiple geographic locations). Datanodes do not know much about the logical files as a whole; instead, they keep metadata about each chunk, including a CRC value for the chunk and identifier information. HDFS also has another role called a Namenode. The Namenode is the master of the HDFS cluster, and contains a database of the file system namespace, mapping logical files to physical blocks in the Datanodes. In some versions of Hadoop, this is a single point of failure, and in any case, it represents the most sensitive part of the HDFS architecture. This database is called the Fsimage. Whenever an HDFS client requests a file or performs other operations, the Namenode orchestrates this request by providing the appropriate mappings, which the Datanodes then service by providing the block and CRC to the client. The Namenode also manages block replication. The Namenode manages changes to this image via a transaction log called the EditLog. This EditLog is periodically justified against the Fsimage. Also, to keep records up-to-date, a Datanode sends a report to the Namenode of all the data blocks it contains on each start-up [2].

2.2.2 Trusted Computing, Trusted Platform Module, and Enabling Tools Overview

The Trusted Computing Group (TCG) is an industry consortium that advocates the use of Trusted Computing technology. It develops the specifications for what it believes are the key enablers of Trusted Systems, including the Trusted Platform Module (TPM) and various protocols such as Trusted Network Connect and IF-MAP. The Trusted Platform Module is designed as a commodity chip that is integrated into motherboards

from Intel and AMD, as well as appliances such as network switches, firewalls, and embedded devices [10]. The TPM provides features that are useful in providing assurances about the state of a platform and protecting sensitive information. The primary design goals of the TPM were to create a low-price hardware root of trusted components that ensures the following: private keys cannot be stolen or given away; the addition of malicious code is detected; malicious code is prevented from using private keys; and encryption keys are not easily available to a physical thief. The TPM achieves this via public key authentication functions, integrity measurement functions, and attestation functions [11]. Essentially, the chip can be used to generate, store, and protect encryption keys. It also provides a mechanism for storing information about the state of a platform through a traceable, cryptographic mechanism that can be securely attested to a remote verifier [12]. In addition, the TPM contains a unique private key that can be used to validate that a platform is indeed what it says it is. On its own, however, the TPM does not do much. It contains no inherent ability to provide security functions; instead, it relies on software that takes advantage of its functionality [10]. Although it provides cryptographic functions, it is not a cryptographic accelerator, and as a commodity chip it is not fast enough to be used as such as cryptographic commands can take 100ms or more to return a result [13]. This fact can limit the extent to which one can rely on the cryptographic functions from within a time-sensitive application. The TPM 1.2 spec also provides a number of monotonic counters, primarily used to prevent replay attacks within the TPM software, and a secure timer that can also be used for this purpose. It also provides additional NVRAM for the storage of additional user-created keys within the chip [14].

The first TPM specification was release in 2001, and since then hundreds of millions of devices have shipped with TPMs. However, likely due to the involvement of Microsoft, the US government, and other large corporate backers, there was a backlash against the technology from privacy advocates and academics, who feared that the devices could be used to enforce DRM and software-use restrictions, and could endanger privacy [15]. Because of this controversy, some valuable uses of the TPM have fallen by the wayside in mainstream adoption. In fact, the concept of attestation via the Privacy Certificate Authority and through Direct Anonymous Attestation has never been implemented commercially. However, with the introduction of Windows 8, Microsoft is using the TPM by default for a measured boot, along with UEFI SecureBoot. This has the potential to increase the adoption of other uses of the TPM and its related software components. In addition, by applying TCG technology to a controlled IT infrastructure, the negative issues regarding privacy are negated, and the technology developed to protect privacy while using a feature like attestation is not needed.

In short, the TPM has the ability to enable several key functions:

- It can securely protect and store keys and data (via binding, sealing, or key storage). Binding is encrypting to a key that is tied to the TPM's storage root key, and data sealing is requiring platform control registers (PCRs) to contain certain values for decryption to take place.
- It can conduct asymmetric key operations on the chip (outside of the view of even a kernel root-kit or other OS-level attack).

- It provides a “smart-card”-like ability to store and protect private keys used in user applications (instead of storing these keys on a disk that could be compromised by the OS).
- It stores hash values in built-in registers called PCRs that represent the state of the software on the system (reset at reboot), and it can attest to the state of the system using these values.

TPM Key Components

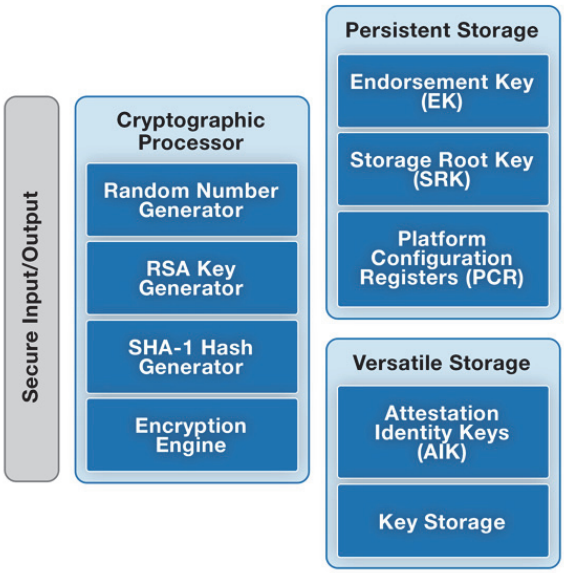

Figure 2. TPM Internals [16]

Cryptographic Processor

The TPM’s cryptographic processor provides the primitives needed to generate keys and encrypt data [16]. These functions are processed wholly within the chip; it is hence impossible to eavesdrop on them from within the host operating system, giving the chip an advantage over software-only solutions. The encryption engine is designed to deal with very low volume encryption (such as encrypting or decrypting other keys).

Since the TPM resides on the LPC bus on a typical system, and the chip is low-cost, a design using it must be aware of bottleneck issues with calling on the TPM too frequently or attempting to encrypt too much data with it.

Platform Configuration Register (PCR)

A Platform Configuration Register (PCR) is an area of memory inside a TPM that is used to store cryptographic hashes of data [10]. The PCRs are 160 bits long and store data from an SHA-1 hash. A PCR is meant to enable the idea of a chain of trust via measurement. As a system is initialized, key components are measured, and their values inserted into a designated PCR. The TPM provides a function with which to extend a PCR with a new value (since there is a limited number of PCRs). For instance, if every piece of code launched by root is measured, each file is measured, and a PCR is extended through a cryptographically secure function. The final PCR value is derived from each piece of code launched.

The TPM 1.2 spec calls for a minimum of 24 160 bit PCRs [5]. PCRs can only be reset at reboot, and are extended as follows: PCR: =SHA-1 (PCR + measurement) [10]. Since the PCR is always extended, it essentially maintains a chain of each item with which it was extended [14]. Once the final application is executed, the PCR should be of a known good value each time, demonstrating that previously loaded components are in a good state. PCRs 0-15 are reserved for Static Root of Trust for Measurement (SRTMs) [10].

Endorsement Key (EK)

The Endorsement Key is a 2048-bit RSA key pair generated on each TPM. It is not changeable after the TPM is manufactured, and identifies the TPM uniquely.

Attestation Identity Key (AIK)

An Attestation Identity Key is a key created for use in attestation protocols. It is tied to a TPM's endorsement key. The idea is that the platform state (PCRs) is signed by the AIK to validate that the PCRs are coming from a particular platform. When this was created, a concept was circulating regarding a Privacy CA that would know about the valid Endorsement Keys; this concept has not gained traction, however, due to privacy concerns with having a central database of knowledge about endorsement keys and about the people to whom they are tied. A new protocol, called Direct Anonymous Attestation, was created as part of the 1.2 specifications to alleviate this concern; still, little software presently uses this mechanism. Attestation can be achieved within an organization by using the Platform Trust Services specifications.

Storage Root Key

The storage root key is an RSA key pair embedded in the Trusted Platform Module (TPM) security hardware. It is used to protect TPM keys created by applications, so that these keys cannot be used without the TPM. For instance, when an RSA key is generated to be used within this Hadoop software implementation, although this key is not physically stored in the TPM, it is wrapped by the TPM storage root key, and hence can only be used on the system with this particular TPM. This means that if data that has been protected by a TPM-generated key is stolen, there would be no way to decrypt that

data on a system that does not have the TPM with the corresponding storage root key. Unlike the endorsement key (which is generally created when the TPM is manufactured), the storage root key is created when a person or organization takes ownership of the TPM. This means that if the TPM is cleared and a new user takes ownership, a new storage root key is created.

Data Binding

Data can be bound (encrypted) to a key that is tied to the TPM's Storage Root Key. This data can only be extracted in the presence of this particular TPM, unless the key is marked as migratable and installed in another TPM.

Data Sealing

Data Sealing requires PCRs to contain certain values in order for decryption to take place. The purpose of this is to confirm that the platform is in a known state before allowing decryption operations to take place. This process gives additional protections against unauthorized system changes or malware exposing access to protected data.

Software Infrastructure for Accessing Hardware Trust Services

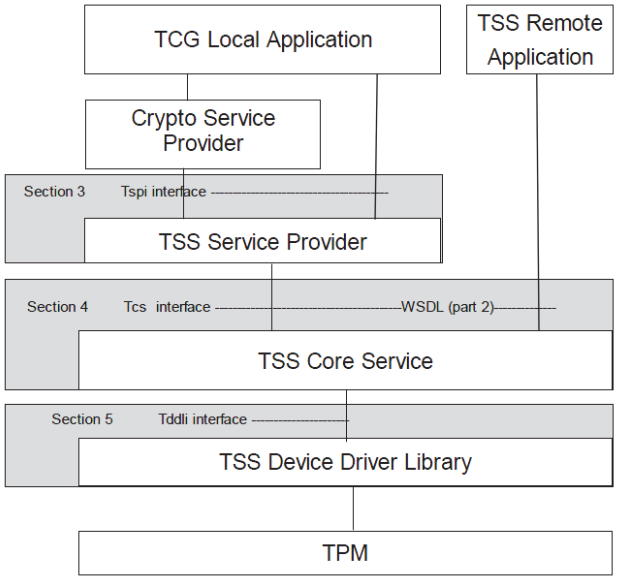

Figure 3. TCG TPM 1.2 Trusted Software Stack Specification

A Trusted Software Stack (TSS) specification provides layers of abstraction to the TPM. This includes a driver provided by the manufacturer API (typically implemented as an OS service), referred to as trusted core services, and an application-layer interface called the service provider that provides a simple interface to TPM functions that other applications can use [8]. Version 1.2 of the TPM spec provides for further ways to abstract TPM access through a remote provider interface (WSDL) or RPC. The TPM 1.2 spec also divides the TSS into kernel services (i.e. device driver) and user mode, which includes user applications and system services. In short, the stack can be summed up as follows: TPM device driver, device driver library, core services, service provider [8] [10].

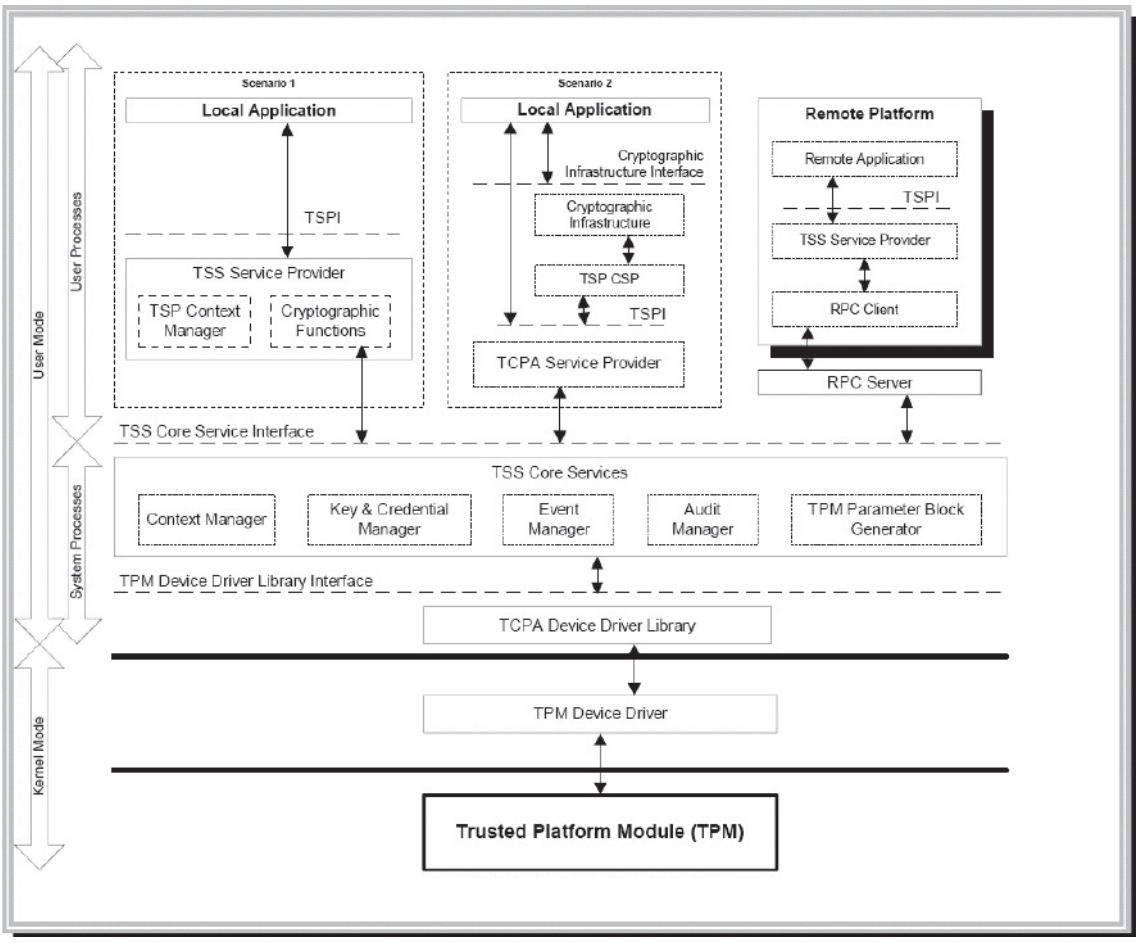

Figure 4. TPM Software Stack [10]

2.2.3 Open Source TCG Tools Leveraged

To implement hardware trust mechanisms in an Apache Hadoop environment, several open-source components that interact with the TPM and related services are used.

Trousers – Trusted Software Stack implemented in C to provide API access to the TPM in Linux. It is used by higher-level Linux-based TPM-related applications such as TPM tools [17]. This tool is leveraged indirectly in the solution as part of configuring the TPM for use, and by higher-level tools.

TPMTools – A software package for Linux that provides access to high-level TPM functions such as TPM ownership, key generation, and PCR access. Accesses to the TPM from an application level can be achieved through the through the Trousers API [9]. This tool is leveraged to set up the TPM and import/export keys, and for general debugging purposes.

TrustedGRUB – TrustedGRUB was used to provide an SRTM (Static Root of Trust for Measurement)-based boot loader for Linux. The Sirrix AG Open Source version was used. It fills PCRs with the values pertinent to the state of the system from the boot-loader level [18]. In addition, a custom checkfile can be provided to measure and extend a PCR with individual files of interest. TrustedGRUB is leveraged as part of a secure Apache Hadoop infrastructure that offers detection of rouge modification of key files, such as the kernel, during system initialization.

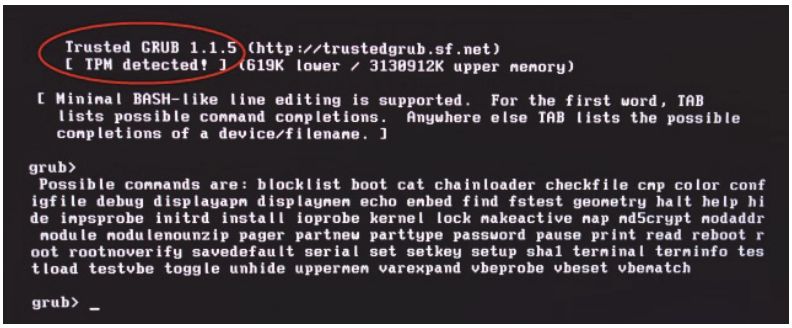

Figure 5. TrustedGRUB

Linux Integrity Subsystem (IMA), Extended Verification Module (EVM) – IMA/EVM provides an integrity measurement and verification service. “The goals of the kernel integrity subsystem are to detect [whether] files have been accidentally or maliciously altered, both remotely and locally, appraise a file's measurement against a ‘good’ value stored as an extended attribute, and enforce local file integrity” [19]. It can: a) collect –

measure a file before it is accessed; b) store – add the measurement to a kernel resident list and, if a hardware Trusted Platform Module (TPM) is present, extend the IMA PCR; c) attest – if present, use the TPM to sign the IMA PCR value, to allow a remote validation of the measurement list; d) appraise – enforce local validation of a measurement against a good value stored in an extended attribute of the file; e) protect – protect a file's security-extended attributes (including appraisal hash) against off-line attack [11]. File integrity measurements are stored as an extended attribute, "security.ima," which EVM can be configured to protect by calculating a hash over the extended attributes in the security namespace (ima, selinux, SMACK64); it uses the TPM to sign it, and stores it as the security.evm attribute on the file [20] [21]. IMA/EVM provides a way to maintain a measurement of the Trusted Computing Base (TCB) accurately by measuring everything opened by root and extending the TCB PCR with this measurement. IMA/EVM was used to store and protect measurements of Trusted Computing Base components and other important configuration files for real-time monitoring of files important to the Hadoop node infrastructure, including Hadoop configuration files and software, providing runtime detection of file compromise.

Open Platform Trust Services – OpenPST is an experimental project based on the Trusted Computing Group's Platform Trust Services standards. OpenPST provides software to attest securely to a platform's state to a remote verifier. The collector and verifier service are both included in the package [22]. OpenPST is leveraged to verify the integrity of nodes within the Hadoop cluster.

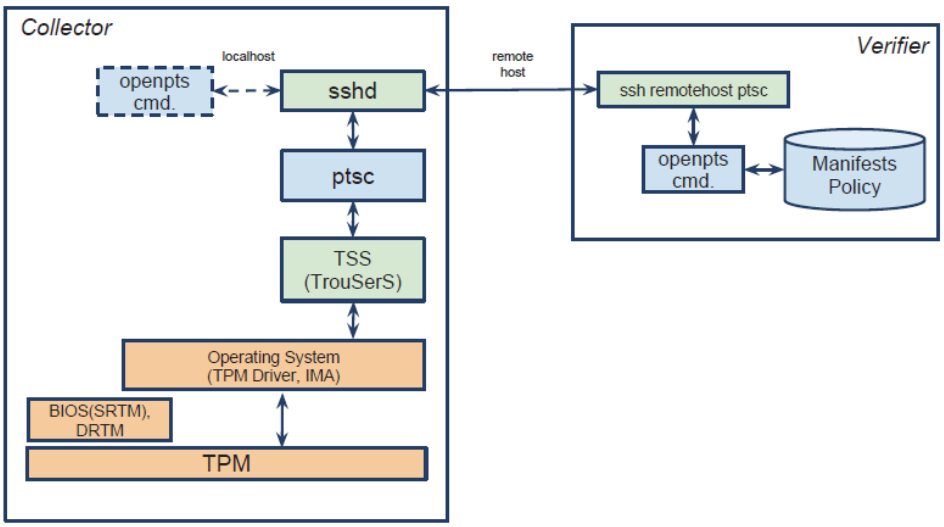

Figure 6. OpenPST Architecture [18]

JTSS *(Java Trusted Software Stack)* – JTSS is a Java implementation of the Trusted Software Stack. It provides a way to access TPM functions through an API implemented in Java. It can work independently or in conjunction with (as a wrapper for) Trousers [23].

> Conforming with the TCG TSS specification, IAIK jTSS consists of two major parts: The TSP and the TCS. The TSP library is the entity that provides application developers with an API that allows access to all the TPM functions. The TSP is designed to be linked to an application that wants to make use of a TPM. The TCS is intended to be the only entity that directly accesses the TPM. As a consequence, the TCS is meant to be implemented as a system service or daemon. It is responsible for creating the TPM command streams, TPM command serialization, TPM resource management, event log management and the system persistent storage. [23]

javax.trustedcomputing – Java Specification Request (JSR) 321 provides a high-level API with which to make use of a TPM within a Java application through a javax.trustedcomputing package [24]. As an API, it requires an implementation of the TCG TSS to function, which is jTSS.JSR32. This goes a long way to making pervasive use of Trusted Computing technologies at the Java application level both feasible and relatively easy. TCG concepts and the concept of ease of use have traditionally been mutually exclusive for a number of reasons. Having a powerful API that allows secure key management and platform integrity measurements opens up a good many interesting possibilities. This still-experimental API was used in this design, in conjunction with jTSS, to facilitate the use of the TPM. A utility class was created to simplify the TPM functions that were required to implement the solution. The code excerpt below illustrates how simply data can be bound to a TPM key:

```
public byte[] bindData(BindingKey bindingKey, byte[]input) throws Exception {
loadStorageRootkey();
javax.trustedcomputing.tpm.tools.Binder binder = context_.getBinder();
byte[] boundData = binder.bind(input, bindingKey.getPublicKey());
return boundData;
}
```

Example 1. Utilizing javax.trustedcomputing for Simplified TPM Access

By applying TCG technology and related open-source components to the application-specific problem of threat mitigation in an Apache Hadoop infrastructure and software design, effective mitigation of threats to an Apache Hadoop environment that are addressed through integrity verification and data encryption with strong cryptographic key protections can be demonstrated. The net result is an infrastructure that can be

measured and determined if it is in a trusted state and a storage platform that is protected against unauthorized access and modification.

2.2 Related Research

As of June 2012, when the current research was initiated, there were no related works of research found on the specific topic of creating a trusted Hadoop platform. In the intervening months, there have been several related publications whose contributions are summarized in the following.

TSHC: Trusted Scheme for Hadoop Cluster

The authors present a framework within which to create a trusted architecture for Hadoop [25]. They extended the inter-node and client communication protocol with built-in attestation, and also build the cluster on a Trusted Computing Base. In this way, node and client communications have a mechanism with which to check the integrity of the services (presumably via checking PCRs for known configuration) as part of the session-initiation process. The paper is a sparse on exact details of how this was accomplished; however, it offers the possibility of encryption based on unique session keys derived from TCB-based public/private key pairs. The concept of inter-node attestation is a topic considered in Chapter 6, and is also addressed indirectly via the use of a generic attestation service in the IT architecture. Overall, it would appear that the goals of this research are complementary. The first paper I published on this topic was cited as a reference in this work.

TMR: Towards a Trusted MapReduce Infrastructure

This work provides an implementation of a purposeful attestation engine in MapReduce, which is also complementary to my efforts and to the concept of a purposeful attestation engine for HDFS [26]. The figure below from this publication highlights the concept of purposeful attestation for MapReduce.

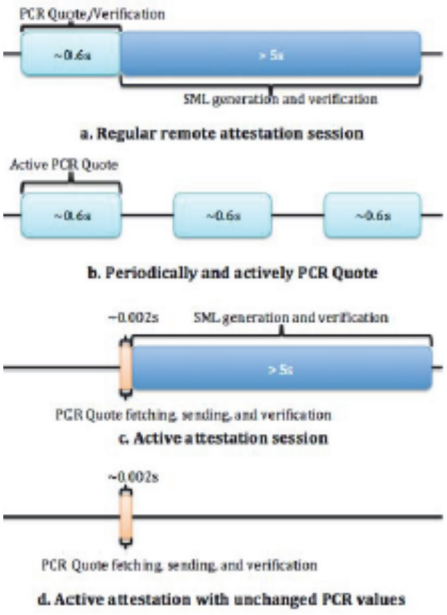

Figure 7. Attestation example with Timings [26]

The primary aim of this research is to develop a measurable, verifiable platform to which to submit MapReduce jobs. To that end, Trusted Computing-based attestation protocols are used. The figure below, also from this publication, illustrates a similar Trusted Computing stack as illustrated in the implementation in Chapter 4 of this dissertation.

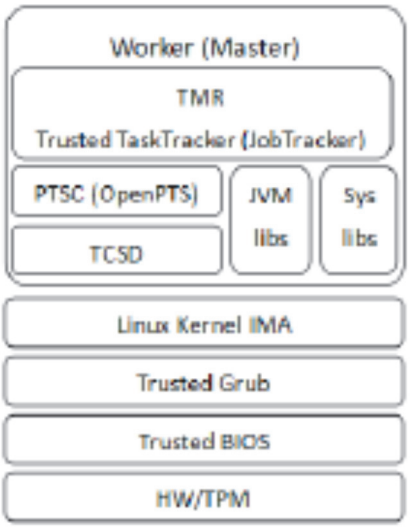

Figure 8. Trusted MapReduce Stack [26]

The paper also presents a study of the performance implications of adding the Trusted Computing-based attestation protocol and additional integrity heartbeats to the MapReduce Task Tracker. Their results yield a very reasonable 1.24 percent overhead. Full attestation completed in five seconds, while PCR quotes and verification completed in 0.659 and 0.0016 seconds respectively.

Secure Hadoop with Encrypted HDFS

In this article, the authors recognize the lack of data-at-rest encryption and implement an AES encryption solution using Intel AES-NI (after also discussing the use of CUDA) [27]. Their results show a seven percent reduction in performance. I also took advantage of AES-NI as part of the software implementation, but did not use the Compression Codec of which this research takes advantage. This work is consistent with the recently (2013) released Intel Project Rhino distribution of Hadoop that offers encryption in a similar fashion. This publication did not address key protection.

Design of a Trusted File System Based on Hadoop

This publication reviews the data-security issues within Hadoop, and proposes a new design for a trusted file system [28]. This system uses fully homomorphic encryption and authentication agents. The homomorphic encryption technology enables the encrypted data to be operable in order to protect the security of the data and the efficiency of the application. Homomorphic encryption is a form of encryption that allows specific types of computations to be carried out on ciphertext and generates an encrypted result that, when decrypted, matches the result of operations performed on the plaintext. The authentication agent technology offers a variety of access control rules that are a combination of access control mechanisms, privilege separation, and security audit mechanisms, to ensure the safety of the data stored in the Hadoop file system.

A Way of Key Management in Cloud Storage Based on Trusted Computing

Although not directly related, this article presents an interesting way of using TPMs combined with an HDFS-based key management scheme that allows users to create “trusted” (i.e., encrypted) storage on a public cloud infrastructure [29]. The concept is for users to have symmetric keys that are protected by a TPM RSA key. These keys can go anywhere, but are useless without the TPM key. The authors offer a framework within which to share the wrapping TPM keys securely via key migration, and hence offer a way to create encrypted storage on public infrastructures that can only be decrypted by users with TPMs with the appropriate wrapping key. The software framework proposed in my research also uses migratable TPM keys among Datanodes, supporting encryption within the HDFS cluster.

In the following Chapter, an analysis of the potential threats facing an Apache Hadoop instantiation will be discussed. These threats and vulnerabilities will illustrate the need for an improved, trusted infrastructure and software changes to improve the security robustness of the Hadoop architecture.

THREAT MODELING

Trust (n): assured reliance on the character, ability, strength, or truth of someone or something. –Webster's Dictionary

In humans, trust is comprised of a value judgment based on available data about the person or thing to be trusted, and perhaps a bit of faith. How does this concept of "trust" translate to the realm of computing, in particular cloud or distributed computing? As is the case with human interactions involving trust, building trusted computer systems cannot always be based on proven models, as complex systems are built upon many components that can total millions of lines of code. As a result, we are left with a value judgment involving a risk/threat analysis of the system, the precautions and protection mechanisms available to limit risk, and what can be proven about the system. I will follow this concept as it relates to creating a "trusted" Apache Hadoop HDFS IT architecture and software platform by beginning with an overview of the potential threats to HDFS.

3.1 Potential Threats to HDFS

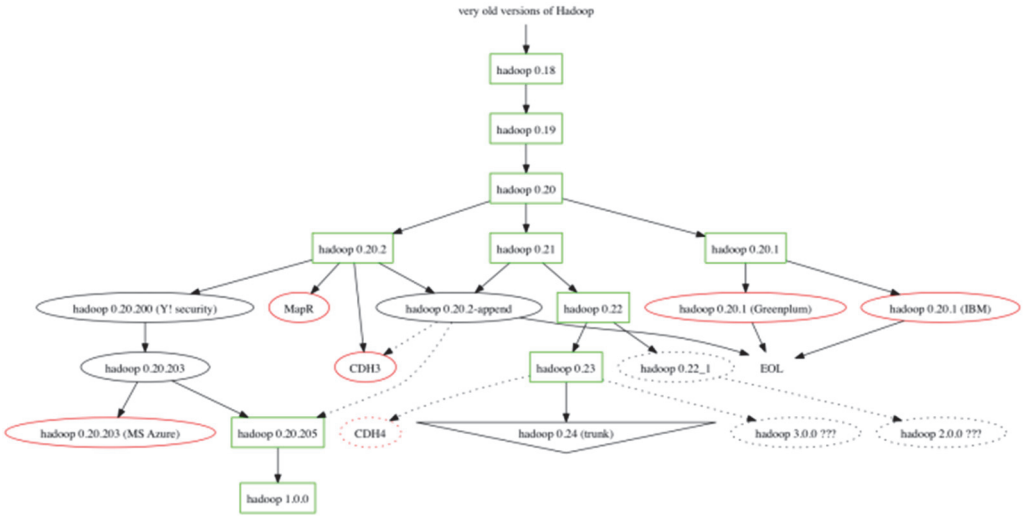

Figure 9 Convoluted Versioning History of Apache Hadoop [30]

Two immediate issues present themselves when determining a threat model for Apache Hadoop. One is the multitude of possible versions that one can use; there exists a number of Hadoop versions in the wild (over 25), each with different features and possible bugs [30]. This is one inherent problem with open-source efforts in general. The Cloudera distribution of Hadoop (with CDH5 being the current version at time of writing) has become a popular bundle of Hadoop for organizations looking to deploy Hadoop in an agile manner, as it includes a number of tools to automate the process and provide training and educational resources. Given the popularity of this distribution, it will be the focus of this model. The second issue is how Hadoop is configured by default. Even with the automation present in the Cloudera packages, there is no automation to configure it easily in its recommended "secure" configuration. Configuring it in such a fashion is not for the timid or amateur system administrator. Lack of security by default and version control gone awry present their own set of external threats to the security of Hadoop. A plethora of versions makes patch management difficult, and an out-of-the-box insecure configuration begs to be left in said configuration by inexperienced administrators. It is clear that the author of Apache Hadoop built the platform to model the important aspects of the Google papers related to distributed file systems: speed and scalability. Understandably, security was a secondary consideration. Cloudera explains: "Even though HDFS has had file and directory permissions since version 0.16, without strong authentication guarantees, these permissions were only useful to prevent accidental data loss. Malicious users could easily impersonate other users, rendering the enforcement of the permissions impossible. Furthermore, even if users could be authenticated to HDFS, all map tasks necessarily ran under a single, shared user account,

which allowed users to access each other's resources" [31]. Security is still playing a catch-up role in Hadoop, mainly in the form of Kerberos integration for user authentication, along with improvements to the code in terms of input validation and stability. In 2009, Yahoo! published a document describing a new security architecture for Hadoop, which includes the use of Kerberos [3]. This document listed the following reasons for the new design:

> Hadoop services do not authenticate users or other services. As result, Hadoop is subject to the following security risks. (a) A user can access an HDFS or MapReduce cluster as any other user. This makes it impossible to enforce access control in an uncooperative environment. For example, file permission checking on HDFS can be easily circumvented. (b) An attacker can masquerade as Hadoop services. For example, user code running on a MapReduce cluster can register itself as a new TaskTracker. (c) Datanodes do not enforce any access control on accesses to its data blocks. This makes it possible for an unauthorized client to read a data block as long as she can supply its block ID. It is also possible for anyone to write arbitrary data blocks to Datanodes. [32]

The document goes on to explain that Kerberos was picked over SSL due to the latter's reduced speed in asymmetric encryption and the increased overhead of PKI management, and that the mere three percent degradation of performance experienced with Kerberos was acceptable. The document also explains how three types of tokens are used, and how true secrets are generated and stored. Although the security additions were an improvement, they did not solve all of the concerns surrounding the use of Hadoop in a sensitive environment. If Hadoop is configured with Kerberos, it becomes more

difficult to attempt user impersonation, as tickets are used to validate users in separate MapReduce jobs; however, there are still some assumptions made about the state of the networks that open up possibilities in the threat model. Cloudera explains: “The security features in CDH4 meet the needs of most Hadoop customers because typically the cluster is accessible only to trusted personnel. In particular, Hadoop's current threat model assumes that users cannot: 1. Have root access to cluster machines. 2. Have root access to shared client machines. 3. Read or modify packets on the network of the cluster” [31].

In addition, there is no support for data-at-rest encryption in the current Hadoop common. However, there is encryption support available via third-party encryption products like Gazzang, hardware appliances, and through the Intel distribution released in February 2013 (project RHINO). An implementation of the Intel framework into the common has been proposed via the Apache issue request HADOOP-9331. Details of this implementation are presented in Chapter 6, as well as potential threats. Until the most recent releases, encryption of data on the wire was not supported [31]. These limitations leave Hadoop vulnerable to a number of attack scenarios, even in situations in which the assumptions of the threat model are applied. In particular, lack of encryption and data-in transit-protections can present issues in outsourcing Hadoop resources to a cloud environment. The recent implementation of HDFS network encryption and RPC encryption requires both custom configuration and the presence of Kerberos.

3.1.1 The STRIDE Model

Property	Threat	Definition
Authentication	**S**poofing	Impersonating something or someone else.
Integrity	**T**ampering	Modifying data or code
Non-repudiation	**R**epudiation	Claiming to have not performed an action.
Confidentiality	**I**nformation Disclosure	Exposing information to someone not authorized to see it
Availability	**D**enial of Service	Deny or degrade service to users
Authorization	**E**levation of Privilege	Gain capabilities without proper authorization

Figure 10. STRIDE Threat Model Definition [4]

When it comes to potential threats to software, Microsoft's Security Development Lifecycle initiative provides a sensible approach. It provides a practical model by which developers, engineers, and business process people alike can consider software security. Because of the nature of the development process in regard to things like the multitude of external components often used, outsourced development, etc., it becomes difficult to apply a single formal method to developing threat models. With this in mind, Microsoft's STRIDE model allows members at all levels of the development process, as well as end users and outside entities, to consider potential threats and possible mitigations. STRIDE refers to six types of software threats: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. Next, we will examine these six types of threats as they relate to HDFS.

Spoofing

Spoofing is possible in the current Hadoop security model in several ways. First, it is still possible, and in fact usual, for Hadoop to be configured without using Kerberos for authentication, leaving it open to known user-account control issues. Hadoop's use of tokens and Kerberos tickets is potentially vulnerable to "pass-the-hash"-style attacks. Although Kerberos has anti-replay mechanisms, a pass-the-hash attack could use the authentication secret (which has renewal intervals ranging from 24 hours to seven days) to create new sessions [33] [34]. In order to conduct such an attack, an attacker would have to observe network traffic or have physical access to clients or servers participating in the infrastructure. An attacker with access to a Namenode or Datanode could also access tokens stored there to impersonate users. As Becherer states, "In the case of the Block Access Token the symmetric key used in the HMAC-SHA1 will need to be distributed to the Namenode and every Data Node in the cluster" [35]. With Kerberos, but not Hadoop itself, spoofing can be achieved by an attacker who compromises a user's credentials via a directed attack. A user's host could be compromised via an unrelated attack, and a keylogger planted to gain access to the user's Hadoop credentials. An authentication mechanism less-dependent on passwords, such as a Kerberos configuration that accepts smart card/PKI and multifactor authentication, could help mitigate this risk.

Tampering

Not including Intel's distribution introducing data encryption or third-party encryption solutions, data is stored in HDFS in the clear. In fact, even with the use of the crypto framework, if care is not taken in the distribution and storage of encryption keys, this protection becomes weak. For instance, a key embedded in a MapReduce job in the

clear could be monitored by an attacker on a node and compromised. The following is a brief overview of how HDFS works, taken from the architecture document published by Apache, and how data could be tampered with at various points within the architecture. HDFS is a virtual file system, written in Java, which distributes chunks of files across Datanodes, with metadata operations and file operation coordination handled by a Namenode [2]. The chunks, called blocks, are stored on the physical file system of each node in a configurable location. Datanodes keep metadata about each chunk, including a CRC value for the chunk and identifier information. Since blocks and metadata are stored unencrypted on the physical storage of the Datanode, an attacker could potentially assemble and alter blocks directly and modify checksums. HDFS also has another role called a Namenode. The Namenode is the master of the HDFS cluster, and contains a database (the FSimage) of the file system namespace and a transaction log (the EditLog), mapping logical files to physical blocks in the Datanodes. Whenever an HDFS client requests a file or performs other operations, the Namenode orchestrates these requests by providing mappings to the appropriate Datanodes containing the blocks [2]. An attacker with access to the Namenode could alter the FSImage to point to new sets of data blocks, or alter metadata about the blocks.

Non-Repudiation

Kerberos adds a stronger authentication mechanism to Hadoop. Without this enabled, spoofing a user is much easier; hence, being able to claim repudiation is reasonable. With the previous issues of token-passing and the ability of an attacker with physical access to HDFS nodes to bypass authentication mechanisms within Hadoop, it is reasonable to say that non-repudiation mechanisms in Hadoop are weak, although such

claims would require a forensic analysis of the environment to validate any claims about non-repudiation.

Information Disclosure

With Hadoop configured with Kerberos and RPC/block network encryption, information disclosure issues are mitigated under normal conditions. However, if network encryption is not being used (as with RPC and data block transfers), information could be captured by an attacker with access to network traffic. In addition, an attacker gaining physical access can access unencrypted data blocks directly from the OS file system. An attacker replacing the un-validated Hadoop code could add a covert exfiltration channel.

Denial of Service

The network listening services on Namenodes and Datanodes could be vulnerable to denial-of-service attacks. The likelihood of this type of attack is generally mitigated due to network isolation practices that are recommended for a Hadoop environment. However, rogue applications, attacks that penetrate the network boundary, and malicious insiders all have the potential to create DoS attacks by flooding the Namenode(s) with requests. The Apache issue report HDFS-945 recognizes this possibility, listing malformed packets, open-socket limits, and RPC flooding as possible attack vectors [36]. The issue-tracking thread mentions several possible solutions, including rate-limiting logic.

Elevation of Privilege

With the addition of Kerberos to Hadoop, elevation-of-privilege attacks become more difficult under normal circumstances. However, the use of delegation tokens by an

HDFS to reduce Kerberos traffic could potentially allow for elevation-of-privilege attacks by interception of delegation tickets on the network and reuse of said tickets to gain privileged access, or by gaining access to a node and intercepting a delegation token. In addition, physical access to a node would allow for access to the unencrypted data blocks outside of any HDFS checks, bypassing the security model.

The Hadoop community, recognizing the need for a more robust authorization framework for enterprise applications, has made strides towards a fine-grained access control framework that works in conjunction with the common Hadoop applications Hive and Impala [37]. The framework, called Sentry, "is an authorization module for Hadoop that provides the granular, role-based authorization required to provide precise levels of access to the right users and applications" [37]. It has support for role-based authorization, fine-grained authorization, and multi-tenant administration, allowing one to build multi-user applications on top of Hive and Impala by segregating access to data sets for appropriate users and delegating the permissions management to local database administrators [37]. Sentry has an extensible architecture, and can be used with other Hadoop-based applications by creating plugins. Organizations implementing Sentry will have better protections than those offered by the course-grained HDFS permissions when using Hive, and hence will have more protection against unauthorized data access and disclosure. Finer-grained access control and limiting the reliance on the course-grained HDFS security framework mean that users limited to Hive access via Sentry will not have the same opportunity for elevation-of-privilege attacks or unintentional access to data to which they should not have access.

3.1.2 Potential Code Vulnerabilities

A casual perusal of the Hadoop codebase, and HDFS in particular, reveals concise, fairly easy-to-understand code. As Hadoop is written in Java, it presents certain security advantages over storage solutions implemented in other languages. Specifically, the Java Virtual Machine (JVM) provides protection against runtime input that could result in buffer overflows in traditional code. The virtual machine isolates the code, and provides bounds checking and a managed memory system [38]. However, the JVM itself is vulnerable, as it translates byte code to machine code. There have been a number of vulnerabilities related to the JVM. Also, with open-source JVMs making their way into systems courtesy of OpenJDK, an advanced intruder could replace the JVM with a tainted version to aid in data exfiltration.

Security within HDFS code in terms of input validation and fuzz testing is being addressed within the community, as can be seen with Apache issue reports like HDFS-3134 and HDFS-3346 (and many others). HDFS-3134 recognizes that input validation issues in the EditLog on HDFS Namenodes) has the potential to create null pointers, out-of-memory conditions, and unchecked exceptions if given malicious input [39]. HDFS-3346 involves implementing a fuzz checker. Activity like this demonstrates an increased interest within the community in tidying up the coding issues related to security. A search within the issues database for the term “malicious” shows 47 issues, with 14 open. The fact that, at time of writing, only three vulnerabilities relating to Hadoop were listed in the National Vulnerability Database is interesting, considering the number of known issues in the authentication scheme [40]. Although this does speak to the quality of coding, it may also speak to a lack of study in the area of security risk in this increasingly

deployed tool. It also indicates a consideration of Hadoop's security when deployed with the expressed limitations and security assumptions, which may not be practical. Another note regarding formal vulnerability disclosure in a 2011 report by DV Labs notes that the decline in commercial vulnerability reporting may be hiding a deeper issue of a growing market for private sharing of vulnerabilities [41].

3.1.2.1 Static Code Analysis Experiment

To get a better quantitative idea of potential coding issues within the HDFS codebase that could lead to security concerns, a scan was conducted of the HDFS-related source code using a commercial static code analysis (SCA) product (HP Fortify). The scan was conducted against the Cloudera CDH4 distribution hadoop-2.0.0-cdh4.1.2. The tool scanned 194,073 lines of code and identified a large number of potential vulnerabilities (1134 high and 170 critical). One issue with SCA is that it has no sense of the context in which the various pieces of code are being used, and each report has to be individually inspected to validate that the issue is a true one. Inspecting several of the reported issues indicated that the true number of relative vulnerabilities revealed by the analysis was likely much lower than indicated. Nevertheless, given 194,073 lines of code, one must assume the presence of defects that could allow a point of entry for an attacker wishing to gain access to HDFS data. Ultimately, if a code exploit is used to access or manipulate HDFS data, many protection measures that will have been put in place at the system level will be invalidated, as will any protection measures added to the code to protect data. For instance, a block encryption option could be bypassed if one were able to use a vulnerability to manipulate the program into decrypting a block.

Issue Count by Category	
Issues by Category	
Privacy Violation	735
System Information Leak	182
Cross-Site Scripting: Reflected	175
Path Manipulation	124
Command Injection	73
Header Manipulation	5
Code Correctness: Regular Expressions Denial of Service	4
Header Manipulation: Cookies	3
Poor Error Handling: Return Inside Finally	2
Insecure SSL: Android Hostname Verification Disabled	1

Table 1. Commercial SCA Results of HDFS Code

Although the distributed file system portion of Hadoop is not the only potentially vulnerable component of a Hadoop infrastructure, this component is where the data is stored, and hence is an ideal point of compromise. Although not specifically addressed in this model, the MapReduce framework also presents a set of potential entry points for unauthorized data access via submission of rouge jobs and software flaws.

3.2 Advanced Persistent Threats

Why talk about Advance Persistent Threats (APT) in the context of Apache Hadoop? "Big data" has become a hot topic in the enterprise space within recent years, perhaps even rivaling the popularity of "cloud" in the buzzword count within popular trade literature. But what does "big data" mean, and what are the implications for an organization's security posture when using popular platforms to store and process this data?

"Big data" is a single term used to describe an array of possible scenarios, meaning different things to different organizations. In general, it means being able to store and reason about large quantities of both structured and unstructured data. For

instance, in the world of a search giant like Google, this means generating near-instant responses to search queries from an enormous amount of raw data from an enormous number of web pages. It also means being able to take information about a user's internet usage and create target advertisements (a scenario now common across the Internet). What projects like Apache Hadoop have done is bring this capability to the masses through non-proprietary, open-source software that can scale to fit practically any organization's needs. The implications of this increasingly popular and affordable solution are that big data analytics are accessible to organizations in any business space.

Advance Persistent Threats (APTs) has become another popular term entering the vernacular from the realm of information security; but what does this mean, and how are these any different from any other threat? The practice of "playing defense" in the field of information security industry is difficult one. Attending an information security event such as Defcon, for example, gives one an appreciation of the fact that, as hard as one may work to defend the software he or she writes or the network he or she protects, someone is always out there looking to break it. The fact is, the attacker has a number of advantages over the defender. The defender's job is equivalent to defending a castle against a horde of invisible, determined invaders who never sleep. The defender has the advantage of being able to see the internal structure of the castle, control the troops within, and set up any number and depth of defenses. These defenses are generally good enough to block the casual attack coming from script kiddies and known worms/viruses. But unfortunately, the defender can only guess at what the enemy is going to throw at him or her, based on known existing vulnerabilities and an assumption that no one on the inside is trying to undermine his or her efforts. He or she also needs to let the "users" in

the castle conduct business while somehow keeping them from opening a door to let the invaders in. Meanwhile, all the invaders have to do is come up with new invasion techniques, find an opening in the defenses, get someone on the inside, or exploit one of the users. They can attempt this on a continual, persistent basis. APTs typically target individuals within an organization through very convincing ploys such as spear-phishing to gain an entry point, and/or may use a previously unknown "zero day" software vulnerability to gain access. Both the RSA security breach of 2011 and Stuxnet (2010) are recent examples. Once inside, the attackers make themselves very difficult to find and remove, setting up multiple back doors, covert exfiltration channels, and diversions. They then proceed to locate sensitive data and find ways around internal protections. Most literature associates APTs with nation-state hackers (i.e., China if one is in the US, or the US if one is in China) or organized crime, which are among the few groups usually able to muster the resources required to come up with zero days; however, so-called "hacktivist" groups and other determined entities or individuals have the potential to become APTs.

For some organizations, the HDFS infrastructure becomes a catch-all for data that has an indeterminate purpose. Given the wide range of applications, it is not particularly hard to imagine that sensitive data could be contained within an HDFS infrastructure, making it a ripe target for an APT seeking a point of long-term exploitation. Depending on the business of the organization targeted, HDFS could make a treasure trove of valuable information vulnerable to a potential adversary. A recent *Forbes* article estimates that the big data industry is worth about $5 billion in factory revenue, with projections increasing to $55 billion by 2017 [42]. The old adage of "follow the money"

applies to cyber targeting: if the organization maintaining the HDFS can gain financial benefit from the data stored within it, someone else may be able to as well. To take an obvious example, a defense organization could use Hadoop to store and process raw intelligence information. Being able to access or alter this type of information, or even being able simply to determine the type of information being stored, would be an extreme benefit to a nation-state adversary. In the context of healthcare, Hadoop could be used to aid in fraud detection within claims and conduct research analytics on medical records, as well as to execute a number of other activities involving personally identifiable information. A breach in this information would be a violation of legal regulations, and damaging to the reputation of the data owner.

Owen O'Malley, one of the chief contributors to the Hadoop security design and Hadoop in general, has stated, "The motivation for adding security to Apache Hadoop actually had little to do with traditional notions of security in defending against hackers since all large Hadoop clusters are behind corporate firewalls that only allow employees access" [43]. However, an attacker specifically targeting this environment, given the right opportunities, could gain unauthorized access to HDFS, even in a "secure" configuration. The key concept in the evaluation of the APTs against Hadoop is that even in a scenario in which a security-conscious organization takes all reasonable precautions to isolate Hadoop services via firewalls, proxies, and monitoring, the APT still has a number of options for entry points and exfiltration channels. This fact indicates that an organization cannot rely on traditional network and isolation protections as the only mechanism through which to protect a standard HDFS environment and develop trust in the platform, as has been suggested in the Hadoop security design [3]. Given the distinct

possibility of sophisticated threats when dealing with sensitive data, more-sophisticated protections are required to establish trust in cases in which sensitive data is involved. The use of encryption with key protection and platform integrity measurements, in conjunction with the measured IT platform that will be described, will increase the level of effort required for exfiltration and data modification attacks and improve the speed at which attacks can be detected.

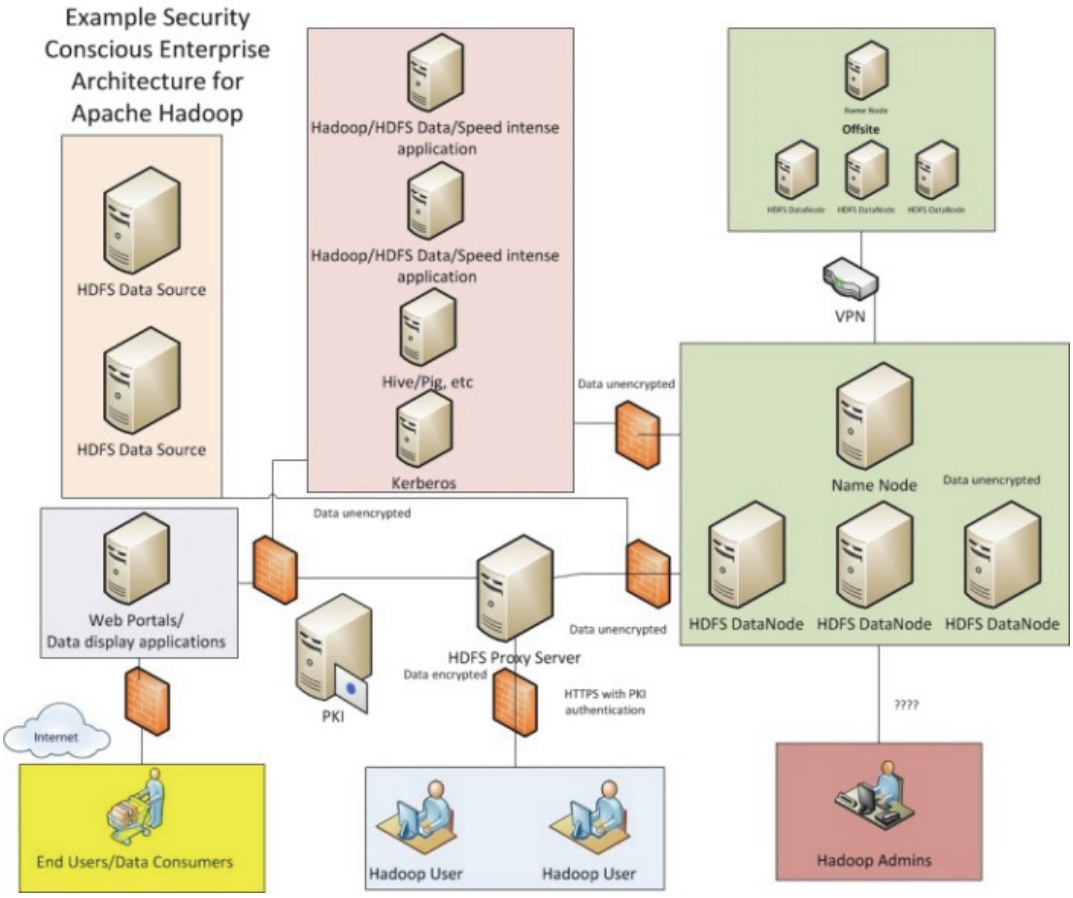

Figure 11. An Example of a Security-Conscious Enterprise Architecture for Hadoop

The diagram above illustrates an example of Hadoop enterprise architecture in an organization that has taken reasonable security precautions to protect the sensitive

elements of the infrastructure. The core idea in this design is to isolate the HDFS data sources with a layered approach. In this way, Hadoop users and applications using Hadoop are connected to the data through firewalls and an HDFS proxy server. Isolating the HDFS core infrastructure of Namenodes and Datanodes in this fashion allows for filtering of incoming traffic to limit authorized sources. For instance, only the related HDFS application servers, Kerberos source, and HDFS proxy server should be allowed to access the infrastructure directly. Also, data going to an offsite facility hosting additional HDFS resources is contained and encrypted in a VPN, alleviating data-in-transit attacks. In this scenario, an end user accessing an application on the Internet that ultimately stores or retrieves data from HDFS is isolated from that data by several boundaries. Along with other security best practices (OS patching, intrusion detection, malware scanners, etc.) and proper configuration, this would be considered a reasonably secure implementation, and the data can be considered safe in most scenarios. This is the type of environment in which Hadoop, even with security settings enabled, is designed to operate. However, the "A" in APT stands for "Advanced," and an attacker specifically targeting this environment, given the right opportunities, could indeed gain unauthorized access to HDFS. APTs, unlike typical attacks, are patient and persistent in planning their entry into the system and their exfiltration. As a report from an incident response company notes, "The APT successfully compromises any target it desires. Conventional information security defenses don't work. The attackers successfully evade anti-virus, network intrusion detection and other best practices. They can even defeat incident responders, remaining undetected inside the target's network, all while their target believes they've been eradicated" [8]. Given the security measures taken in the diagram, scenarios are

presented showing how APTs can use known techniques and weaknesses in HDFS to extract or compromise data.

APTs can be thought of as having a seven-step exploitation life cycle that includes reconnaissance, initial intrusion, establishing a backdoor, obtaining credentials, installing utilities, privileging escalation/lateral movement/data exfiltration, and maintaining persistence [8]. Outside attackers looking to access HDFS data in the given Hadoop example need to make initial inroads into the corporate network. They have several opportunities to do this. First, they can look for a traditional entry point in front-facing services. The web portal application shown would be a first target, where they would use a traditional vulnerability scanner to find any open ports (management interfaces such as RDP/SSH) or exploitable services. Good firewall policy should block any incoming connection other than http/https from hitting the web portal application. The attackers will identify the web server (i.e. Apache HTTPd, JBoss, etc), and examine it for known issues (or a zero-day exploit) that could be used to obtain a remote shell. They will also look for any unvalidated web input that could be used to execute a command. Aside from seeking access to potential zero-day vulnerabilities in the server software, APT-style attackers look for these easy entry points. If they are able to gain access to the web server with a remote shell, they can use this as a starting point into the network. APTs will quickly look for another place to set up a back door, in case they should be detected. From the portal, they may manipulate the application code to pull data from HDFS, or will move on to compromise the HDFS proxy server. If there is no exploit available, they will target user credentials, perhaps from an admin who accesses the web server. If they fail to gain access through the front-facing services, they will

move on to a social engineering attack. These attacks are aimed at getting credentials for or access to a computer within the organization's network. The attackers will target a particular individual (preferably someone with potential administrative access) with a spear-phishing attack in which the person will either disclose credentials or will be directed to a website that installs malware, providing a backdoor. Attackers may be able to use a zero-day exploit to avoid detection by anti-virus software and network filters.

In this diagram, Hadoop administrators have direct access to the protected HDFS infrastructure. This may be via a VPN, SSH, or some other network segregation. An ideal scenario for the attackers may be to infect this administrator's computer with malware, and use it as a bridge directly into the HDFS infrastructure via the person's administrative channel. The Hadoop admin, for example, may give a presentation at a conference, and be given a USB stick containing malware as a free giveaway. APTs have the resources to conduct operations like this in order to gain access to hard targets.

In any case, once in, the attackers look to protect their connection via establishment of other points of entry and back doors. APTs are knows to set up easy-to-find back doors as well as to give security administrators a false sense that they have solved a detected problem [8]. To establish other backdoors, the attackers may first need to gain more privileged access via pass-the-hash attacks or malware such as key loggers. Their channels in and out of the network are made to appear like legitimate traffic through the use of permitted ports, encryption, or mimicking permitted traffic to permitted sites [8]. Exfiltration data can be hidden in a number of ways, including in plain sight by being mixed into protocols (an example of covert channels). The attackers will explore the internal network to identify possible points of entry into sensitive data

sources, such as HDFS. In the process, they may install various tools to help gain passwords and achieve additional points of compromise.

At this point, the attackers have several options to gain access to the HDFS data. A straightforward approach would be to gain the Kerberos credentials for a service account or individual with access to the data. The issue with this approach is that, unless they find an account with super-user access to all files, the attackers will be limited by HDFS security to the set of files that the user account has access. Compromising the various Hadoop applications shown would have a similar effect. The attackers could monitor and exfiltrate data to which the applications have access. If HDFS traffic is in the clear, they could do this by simply monitoring the network traffic. Compromising the HDFS proxy would be a more ideal target, and could give the attackers better access to the full contents of HDFS. Using open channels through the firewalls protecting the core HDFS services by either exploiting the network applications or via an administrative channel would allow the attackers to manipulate the HDFS environment directly. The symmetric encryption key used in token protection is distributed to each node and stored on the file system. Accessing this key would allow further compromise by creating delegation tokens to access restricted files. Likewise, with physical access to the Datanodes, the unencrypted blocks could be exfiltrated or modified.

For long-term persistence not requiring continued physical presence, attackers could craft a replacement HDFS software package that relayed data streams to another location as they are created or accessed. An attack like this would better-fit the profile of an APT rather than a get-the-data-and-run effort. Again, the firewalls shown continue to do their job, but the attackers use the very ports that the firewall allows them to traverse

in which to hide the data. Even a Layer 7 firewall might not be able to detect the behavior if the data is sufficiently hidden in the protocol or made to look like expected traffic.

A type of less-sophisticated attack could potentially come from an insider threat such as an administrator. With the aim of modifying records, someone with administrative credentials on the system could modify block files and checksum files, or copy the unencrypted block files and metadata database from the Namenode and reassemble the files elsewhere.

These scenarios are possible, and are not trivial. An organization deploying effective security training and policies to its end users and maintaining vigilance through constant monitoring for anomalies can, however, go a long way towards inhibiting these attacks. In addition, by integrating trusted computing principles into HDFS at both the infrastructure and software layers, one arrives at a more robust architecture that will provide another set of obstacles to attackers. Ultimately, being able to slow down the attacks, if not prevent them altogether, will give more time for anomaly detection and systems monitoring.

3.3 Threat Mitigations Based on Trusted Computing

The use of trusted computing concepts in Hadoop and the implementation of HDFS block encryption with TPM key protection does not address all of the threats to a Hadoop architecture. The key protections provided are in the areas of integrity and confidentiality of data. This comes from protection of data against modification outside of the Hadoop software and protection of data from unauthorized access and modification. This also helps to ensure that the protection mechanisms that exist in the

Hadoop software and configuration are maintained and protected against unauthorized modification. The following chapter will explore the design of a trusted IT infrastructure for Hadoop and the design of an AES based encryption scheme for Hadoop with hardware rooted key protections.

THEORY - DESIGNING A TRUSTED ARCHITECTURE AND ENCRYPTION SCHEME WITH TPM KEY PROTECTION TO MITIGATE INTEGRITY, CONFIDENTIALITY, AND ADVANCED PERSISTENT THREATS

Now, having a motivation rooted in realistic usage scenarios and knowing the key vulnerabilities, the following will describe how an IT architecture and HDFS software changes were created to mitigate some of the key vulnerabilities in Hadoop; particularly those related to data integrity and confidentiality.

4.1 A Trusted Infrastructure

My method for developing a trusted HDFS platform begins with the infrastructure. Essentially, this involves implementing key features of the TCG solution set to build a base environment that is measured and verified. This is key to a higher level of assurance, as without establishing this base infrastructure, any additions to the Hadoop software layer would still be subject to many of the traditional operating system security concerns against which a trusted computing base tries to offer protection. These concerns include, for instance, replacement of the Hadoop code or system libraries that would circumvent any protections that have been added to the software. The following is a summary of how off-the-shelf TCG components would be used as part of a Trusted Storage Design, with the goal being to have the base OS platforms hosting the Hadoop code in a known, verified state at launch.

Trust in the TCG framework is built from the hardware up. Although there has been research in the area of providing trusted storage and services on untrusted platforms, being able to start at the lowest layer, while increasing infrastructure complexity, will also provide better assurances at the software level. The idea is to ensure that each component of the operating environment that is relevant to the security and integrity of

the system is measured and verified against known values [12]. Without this root of trust, one cannot be sure that higher-level elements (such as an application, file storage, etc.) have not been compromised at a lower level (such as an operating system root kit). By implementing a measured and verified system launch, when Hadoop code is executed, it will be possible to know if the host operating system, key system files, and the Hadoop software itself are in a known, good state. This idea of a ground-up chain of trust is not a new concept; however, its integration into the mainstream could be described as slow at best. Momentum around hardware-based trust is increasing, particularly with the release of Windows 8, which will implement a "secure boot" and "measured boot" feature to provide protection against root kits and changes to key operating system files [44]. Other technologies such as Bitlocker, self-encrypting drives, and third party security products are already using the TPM to protect encryption keys. Even so, existing software in the Linux environment, the typical platform for Hadoop, can still be considered to be evolving-but-available, and underutilized.

4.1.1 Overview of IT components

Each node of the Hadoop cluster will implement hardware trust components to increase the trustworthiness of the base operating system.

- Each node will implement a chain of trust using open source components to take measurements of key components of the operating system and associated sensitive files. Specifically, each node will implement TrustedGRUB to verify key files, and extend Platform Control Registers with platform integrity information.

- Each node will implement file-level integrity measurements protected by hardware trust. Specifically, implementation of IMA/EVM will augment the measurements from TrustedGRUB.
- A remote validation server will confirm the state of each node in the Hadoop cluster via a remote attestation protocol. Specifically, this will be an implementation of an Open Platform Trust Services collector on each node, and a central verifier. Each node will attest to its status using a secure protocol.
- Each node will have enhanced physical storage of file blocks within an HDFS Datanode and metadata within a NameNode. Encrypted file storage will be implemented using dm-crypt (or encryptfs) encryption, with the key protected by the TPM and a known platform state. Data will not be decryptable unless the system meets some known integrity values determined during startup. Optionally, TCG-compliant OPAL-complaint Self Encrypting Drives could be used to protect offline data, with no overhead, to protect against physical drive theft. Note that although encrypting HDFS data in this fashion was tested, there were a number of limitations that lead to the concept of encryption being built directly into the Hadoop code.
- Data in transit protection using IPSec Tunnels between Hadoop nodes will be implemented. IKE initialization keys will be protected with TPM. The rationale for establishing IPSec tunnels as opposed to using the SASL-based encryption that is now present in Hadoop is that this option was not available during my initial research. Also, the IPsec channels provide the added benefit of encrypting all

traffic among nodes, regardless of whether or not SASL data transfers and RPC calls are enabled within Hadoop.

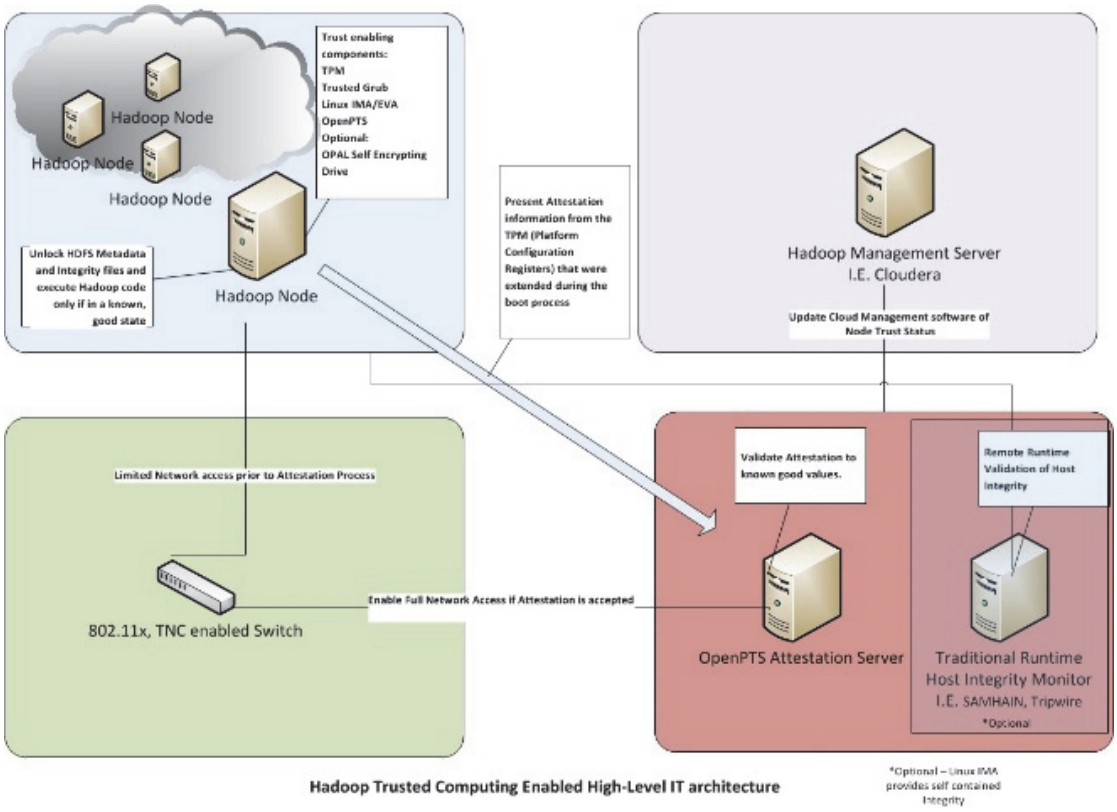

Figure 12. Hadoop Trusted Computing-Enabled High-Level IT Architecture

The diagram above illustrates the components for a Trusted Computing-enabled IT architecture for Hadoop, using a combination of node-level services, network services, and software integration. Two optional components are shown above, the first being a traditional runtime host integrity monitor, which is not related to the trusted computing components. The reasoning here is that although validation of integrity during system initialization is provided, this component can provide an additional level of traditional insight into runtime changes. This software would be part of the measured, trusted

computing base. The second optional component is The Trusted Network Connect (TNC) component. This is only optional because at this time the TNC protocol and supporting software (including OpenPST) are still in the developmental stage. It would not be practical to assume that a TNC-enabled switch is present, particularly due to spotty vendor support, with Cisco currently absent. Also, the real "magic" of TNC is best realized with the IF-MAP framework to integrate other security metadata into access controls, and it may thus be more appropriate for regulating client access to a network with a Hadoop cluster containing sensitive data (see the TCG IF-MAP specification). Integration with TNC could offer the ability to ensure that a Datanode failing to meet integrity requirements would not be allowed to connect to the cluster, or even to the network in general (preventing further possible corruption).

4.1.2 Node-Level Components (OS)

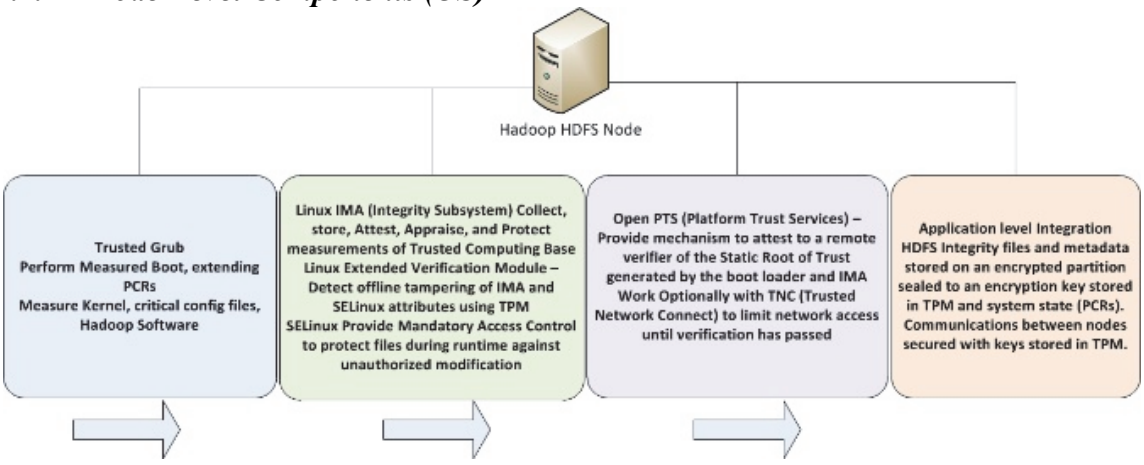

Figure 13. HDFS Node-Level Trust Services

A "chain of trust" is one of the core concepts of Trusted Computing, and a key to providing assurances that the platform is in a known good state and has not been subverted [12]. The general idea is that an entity (say a boot loader) measures the next

piece of code in the execution chain (say the kernel); this entity then measures the next piece of code to be executed (an application), etc. This chain of trust needs an initialization point that is inherently trusted. This is ideally the BIOS boot block. This boot block code would be considered the "Core Root of Trust for Measurement." This type of chain of trust is considered a "static root of trust" (8).

Again, when talking about trust in the IT architecture of Hadoop, we will start from the ground up, so to speak, in establishing a trusted environment. In this part of the Hadoop infrastructure discussion, the focus is on establishing a known, measured operating system for each of the nodes in the Hadoop cluster. There are two options for establishment of the initial chain of trust. The first is implementing a complete static-root-of-trust measurement, which, through a combination of components, is essentially what is defined. TPM-enabled BIOSes are measured as part of a root of trust, and are perhaps an under-considered place to start when examining the security posture of a system. This "firmware" is software as well, and represents another possible intrusion point into a system, particularly with BIOS code base becoming more complex in UEFI-based systems. Industry has recently taken note of the potential for firmware compromises, particularly in response to NIST SP 800-147, published in 2011, which defines standards for BIOS protections. Although BIOS protections are not directly correlated to this protection scheme, it is important to note that proper SRTM starts at this level. To reduce the complexity of the SRTM model, an alternative approach has been developed, called Dynamic Root of Trust (DRTM), using special hardware features such as the Intel Trusted Execution Environment [45]. Instead of relying on a static root of trust from BIOS, this technology makes use of protected CPU operations and dedicated

PCRs while executing certain protected code [17]. The Linux TBOOT project makes use of this technology. Although DRTM has the promise of reducing the complexity and measurements required to execute measured code, so far the practical implementations have been mainly associated with TBOOT. There has also been a demonstrated attack against Intel's Trusted Execution Technology (TXT) using Bios System Management Mode SMM [46].

The OS loader will be considered the first practical step in the chain of trust in this example, beyond the system firmware. TrustedGRUB is a Linux Boot Loader that can use the idea of a Static Root of Trust Measurement [18]. Once a system BIOS starts, PCRs are extended with the values of BIOS components, option ROMS, and the bootloader [10]. The TPM 1.2 spec calls for 24 160 bit PCRs [10]. PCRs can only be reset at reboot, and are extended as follows: PCR:=SHA1(PCR + measurement) [5]. Since the PCR is always extended, it essentially maintains a chain consisting of each item with which it was extended [5]. When the final application is executed, the PCR should be of a known good value each time, demonstrating that previously loaded components are in a good state. The PCR index values (0-24) are reserved for different purposes according to the TCG spec (see TCG 1.2 Main Specification). PCRs 0–15 are reserved for Static Root of Trust Measurements [5]. There is also an idea of trust locality that is applied to PCRs 17–20 and is used with Dynamic Root of Trust Measurement (DRTM), which uses hardware such as TXT [5]. PCR 21–22 are controlled by a "Trusted OS," and PCR 23 is application-specific [5]. After BIOS passes control to the boot loader, the second stage of the measured and verified boot process begins. TrustedGRUB validates each stage of the boot loader and extends PCRs, including the initrd (initial RAM disk,

including the system kernel), before passing control to the kernel [18]. The PCR locations used include the following: PCR 4 contains MBR information and stage1. PCR 8 contains the bootloader information stage2 part1 and PCR 9 contains bootloader information stage2 part2. PCR 12 contains all command line arguments from menu.lst, and those entered in the shell. PCR 13 contain all files checked via the checkfile-routine. PCR 14 contains all files that are actually loaded (e.g., Linux kernel, initrd, modules, etc.) [10]. TrustedGRUB can also be configured to validate other arbitrary files such as the password file. At a minimum, it is prudent to consider the additional measurement of the Hadoop configuration files, Hadoop software packages, and Java software as targets to check with the checkfile. Note that TrustedGRUB does not protect the known good values of files to be checked, which means that if only TrustedGRUB were used, something like an encrypted partition should be tied to the values of the PCRs (to indicate that files have been checked and are in a known good state) and/or should remotely attest the PCR values before letting the node join the cluster. Also, TrustedGRUB does not do anything itself to secure the remaining boot process (a weakness that is being addressed in TBOOT). The next step is to use these measurements as part of a decryption process to decrypt sensitive data that would have been sealed with the TPM's Storage Root Key and bound to these values.

It may be considered sufficient to end the static root of trust at this point. It is possible that measuring the boot loader, kernel, and key configuration files may be considered an adequate check when combined with traditional run-time security such as file integrity monitoring (i.e., Tripwire) and traditional security features to confirm that the system is in a trusted state. The Hadoop HDFS-file system integrity files could be

sealed and bound to the PCRs generated in this state, which would prevent that physical file system from being decrypted if something changes (note that currently Datanode metadata [checksums] are not separable from the directories in which data lives, making this solution more challenging). However, to get more granular control and remote reporting, additional layers are required. This is where the next node-level component comes in—the Linux Integrity Subsystem and Extended Verification Module (IMA/EVM). “Part of the TCG requirement is that all Trusted Computing Base (TCB) files be measured, and re-measured if the file has changed, before reading/executing the file“ [19]. The TCB by definition is all files critical in establishing a trusted environment [11)]. The IMA subsystem enables this by storing and maintaining an integrity measurement log, extending a PCR (PCR 10 by default), attesting (by signing the PCR with the TPM endorsement key), and storing an integrity checksum as an extended file attribute [11]. It can also enable local validation of files. The IMA security attributes process can then be protected by the Extended Verification Module. “EVM provides a framework, and two methods for detecting offline tampering of the security extended attributes [11]. The initial method maintains an HMAC-SHA1 across a set of security extended attributes, storing the HMAC as the extended attribute 'security.evm' [21]. The other method is based on a digital signature of the security extended attributes hash. To verify the integrity of an extended attribute, EVM re-calculates either the HMAC or the hash, and compares it with the version stored in 'security.evm' [13]”.

With the ability to maintain a locally validated base environment, the combination of a trusted boot loader and IMA/EVM will provide a good foundation for a trusted environment in which to launch an application such as Hadoop. Furthermore, IMA

provides a more dynamic run-time verification environment than TrustedGRUB alone, as individual files do not have to be called out for verification. Still, it would not be practical to apply this type of measurement directly to Hadoop data blocks, as it would be normal for these to change, and the extra overhead would be undesirable. The next level in the establishment of a trusted infrastructure for Hadoop will involve remote attestation of the node to a remote verifier. The Trusted Computing Group has defined an interface called Platform Trust Services to define parameters for this exchange between a client and a remote verifier [22]. OpenPST is an implementation of this standard. Platform Trust Services are intended to work in conjunction with other TCG efforts, particularly in the way of trusted network access via TNC. OpenPST has the experimental ability to provide this connection [14].

The idea with platform attestation is to create a reference manifest on a trusted system—essentially a database of known good values for a system—and store this on the remote verifier. An untrusted client will submit an integrity report to the remote verifier, which it will check against the known values in the reference manifest [12]. This process is conducted over a SSH-encrypted channel. In and of itself, this process does not necessarily provide any additional security, however. The next step is to build integration glue. This is the point at which integrating node connectivity into TNC would come in, if available, to block network access. In developing a trusted infrastructure for Hadoop, these results should be integrated into the Hadoop architecture—meaning execution of Hadoop on this node should be blocked, and a management server should be informed that this node is potentially compromised.

Although not part of the hardware based trusted-computing components, SELinux is also shown in the node-level security framework. SELinux provides a policy-based Mandatory Access Control (MAC) framework for Linux as part of the kernel Linux Security Module SELinux. This works in conjunction with the integrity monitoring and validation controls to protect key files from runtime modification [19]. This can, and should, include the actual data block files. SELinux can provide fine-grained access control, and can be configured to allow only the Hadoop Java process to access these files and block other processes and users. One weakness of the proposed model without the addition of a runtime protection like SELinux is the possibility of a system being launched in an expected and verified manner, but having something such as Hadoop data blocks be modified at runtime, since these cannot be controlled with IMA due to the nature of their use. Although considered cumbersome, as shown by the fact that SELinux must be disabled for some Cloudera Hadoop software to be installed, a tuned SELinux policy can give fine-grained control over what users and processes can access and modify, making it more difficult for unauthorized processes or users to modify key files during runtime.

Finally, now that a base system has been established that has been measured from the ground up and attested to a remote verifier, Hadoop software and the HDFS environment can be executed with an assurance that the software and configuration are in a good state, and that the underlying system has not been compromised. The measurements tell us that the HDFS code, for instance, has not been altered to exfiltrate data. Furthermore, using the measurements and ability that have been gained from the implementation of the trusted computing components, value can be added to the Hadoop

application in a few ways. With HDFS, the obvious choices are protecting the Namenode's metadata by binding and sealing via encryption keys and platform state as reported by the PCRs. On the Datanodes, sealing and binding the block integrity metadata can ensure that the data will not be altered when a node is offline. Data in transit among the nodes can be protected by configuring Strong Swan IPSec tunnels using keys protected by the TPM. Also, it is possible that the Hadoop web interfaces could be protected via SSL, with the private key stored on the TPM using openssl-tpm-engine.

4.1.3 Experimental Implementation of Trusted Infrastructure Components and Observed Security Improvements

With the goal of establishing the trustworthiness of the base operating system, a Hadoop environment was created in which each node of the Hadoop cluster creates a chain of trust using the TPM and open source components to take measurements of key components of the operating system and associated sensitive files. This environment consisted of a two-node cluster implementing TrustedGRUB, IMA/EVM, and OpenPTS, as well as encryption and IPSec key protection using the TPM running on Fedora 16. Although architectures implementing TCG Trusted Computing components harbor the promise of increased security through a strong hardware-based root of trust, there are still issues impeding full realization of this promise. Complexity of configuration, and the lack of unity of and support for the various packages implementing TCG capability, stand in the way of mainstream adoption. My implementation of Trusted Computing components to secure a Hadoop HDFS architecture has illustrated how these issues will be roadblocks to an institution looking to implement this solution. For instance,

supporting EVM/IMA in Fedora 16 required rebuilding the kernel from source. In fact, most components required manipulation of the build environment and intense tweaking.

Despite implementation challenges, it was possible to secure data within HDFS in several ways. Each node implemented file-level integrity measurements protected by hardware trust—specifically, implementation of the Linux Integrity Subsystem (IMA) and Extended Validation Module (EVM)—also augmenting the post-boot measurements of TrustedGRUB. The idea of these boiler-plate TCG approaches is to give insight into the integrity of the system at boot time and runtime via IMA. By using a boot loader that measures key operating system files and extends the platform control registers, a decision can be made whether to unlock critical files (such as configuration files, key files, etc.) that are needed for Hadoop. Also, measuring the Hadoop software can be one of these measurements, indicating whether this software was altered. During the experiment, for instance, changes made to the software that was included in the measurements, such as a Hadoop Java package, were easily detectable. In addition, IMA/EVM provides an access log and PCR value for all files that are part of the Trusted Computing Base (which is essentially all files opened by root) [19]. Under normal operational circumstances, this PCR value should be consistent, and checking this value would give insight into possible compromises. In the APT scenario in which an attacker gains access to a Hadoop node directly, he will eventually be detected if the IMA PCR value is checked. Should the attacker modify OS files, Hadoop configuration files, or Hadoop software, this will be detected as well, either through IMA or during the next boot. After the node boots, a remote validation server will confirm the state of each node in the Hadoop cluster via a remote attestation protocol. In the APT scenario, this protection measure will give insight

into the state of the platform, and hence aid in detection of compromise. If the TCB IMA PCR is checked as part of this, then a runtime compromise can be detected; otherwise, the checked values are only valid when they are created at system boot.

Next, some HDFS-specific protections can be applied. First, without these components in place, it is possible to extract information about HDFS-file system blocks and alter these at a manual level via the local file system within an HDFS node. To avoid checksum failure, this would have to be done on any copies of the file within the cluster. Checksums are stored as binary files by HDFS nodes, along with the data chunks; however, these can be updated with the HDFS API by an attacker who modifies the associated chunks. Although this solution does not prevent runtime modification of these files on its own, it is possible to prevent this from occurring when the system is offline because the checksums are in an encrypted directory. The desire here was to seal and bind the checksum data separate from the actual data blocks; this proved difficult, however, as the data blocks are stored in the same directory. As a result, at a system level, a custom script would be needed to replace these with symlinks and move the actual checksum files to a directory encrypted with a TPM key and bound to a trusted PCR state. With such things in place, it was not possible to tamper with data blocks on an offline system without being detected by a checksum failure, as these checksums were encrypted at rest. To reduce the risk of runtime modification, it would be advantageous for the checksums to be individually encrypted and decrypted by the software itself as needed; that way, an attacker would not have access to the checksums simply by having access to the file system, as in the case of a directory that is unlocked at boot time.

Instead, an attacker would either have to manipulate the software to encrypt replacement checksums or gain access to the ability to encrypt a file with the TPM-stored key.

The Namenode partition was also protected with a TPM key and tied to a platform PCR state via dm-crypt [47]. As a result, offline tampering with file metadata was not possible, as the data was encrypted and not accessible outside of the running system. Again, runtime manipulation by a super user would still be possible. Data in transit protections were possible by generating key pairs for use in IPsec using Strong Swan; however, a more automated method of setting up the tunnels with the keys would be desirable. One solution would be to have the private key stored on an encrypted partition that is only unlocked when the system is in a trusted state. Again, this leaves the key open to runtime compromise. Being able to keep the key on the TPM (such as with the OpenSSL TPM engine) without human interaction would be desirable.

4.2 Software Integration

With a measured and verified infrastructure established, I looked at ways to add features to the HDFS application code to improve data confidentiality and integrity. The following are some software integration concepts that were considered, based on the gaps remaining after the enhanced infrastructure components were deployed.

4.2.1 Trusted Computing Software Integration Concepts

Data Integrity

The HDFS client software implements checksum checking on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block

and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents, it verifies that the data it received matches the checksum stored in the associated checksum file [2].

Hidden integrity files could be encrypted and sealed with a TPM key within the application, to increase tamper evidence within the file system. This could be achieved within the HDFS code, or by altering the code to store the checksum files in a separate space that would be bound and sealed at the OS level. If data were altered outside of HDFS, the checksum file could not be altered without accessing the TPM. There is a possible issue with TPM speed in data retrieval and concurrency, but this may be mitigated by the fact that the file chunks are generally large, and the unsealing of the integrity file could take place at the same time as retrieval. Another mitigation could be the creation of a secure checksum and a standard one. The secure checksum would not necessarily be checked on each access, but at random, by request, or during an integrity verification operation when the node is not busy. Another alternative would be to use a secondary process to verify and encrypt file hashes periodically and maintain a database of these hashes as a single file encrypted by a key unique to each Datanode and protected by the TPM. A client application could request validation of the integrity information, if the application has sensitive data or on a random basis. Periodic tamper checks could be conducted on the system against this database of checksums.

Metadata

The FsImage and the EditLog are central data structures of HDFS in a Namenode. A corruption of these files can cause the HDFS instance to be non-functional [2].

The FSImage and EditLog are secured by binding and sealing them to a TPM-protected key and platform state. To accommodate backup Namenodes, this would have to use migratable keys. Periodically, on a clean shutdown, a hash could be taken of the FSImage and sealed with TPM (compared on system initialization). One could potentially use a monotonic counter in the EditLog entries to mark the order of commits. To verify that extra data was not added when a system was offline, an encrypted chain of hashes representing a line in the log could be maintained.

Data Blocks

Typical HDFS applications write their data only once, but they read it one or more times and require these reads to be satisfied at streaming speeds. If possible, each chunk will reside on a different Datanode [2].

The Java framework could manage the encryption/decryption of the data block files, making calls to the TPM directly via a Java Trusted Software Stack (jTSS) interface to release a storage key. This would have the advantage of working outside of the operating system configuration. The potential issue here, however, becomes software decryption overhead. A resolution for this would be that a user could mark more sensitive data as encrypted, instead of encrypting everything. Also, not all data is necessarily needed at streaming speeds, depending on how it is used.

Staging

A client request to create a file does not reach the Namenode immediately. In fact, initially the HDFS client caches the file data in a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates

data worth over one HDFS block size, the client contacts the Namenode. The Namenode inserts the file name into the file system hierarchy and allocates a data block for it. The Namenode responds to the client request with the identity of the Datanode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified Datanode. When a file is closed, the remaining unflushed data in the temporary local file is transferred to the Datanode. The client then tells the Namenode that the file is closed. At this point, the Namenode commits the file creation operation to a persistent store. Blocks are transferred in the clear.

This staged file could be used as a place to encrypt the file blocks, using a key before transmittal. The key could be a locally generated AES key, using PKI (with the key stored or protected by the TPM) to relay the AES key. Alternately, one could use a TPM key marked as migratable between clients and nodes acting in the cluster. Clients would have to request a key beforehand. One could also use a trusted path such as SSL or SCP when transferring the data to the Datanode, or use the TPM to protect trusted path-keying material.

Data Authorization and Protections

HDFS uses UNIX-style permissions on files. A client is authenticated with a username and password. There is optional support for Kerberos for stronger authentication services [2].

This could be extended by incorporating a user identity based on PKI, with interactions with the system over TLS. The user could also be required to access the

cluster from an authorized system, based on a TPM identity. The TPM could be used to store and restrict access to PKI material.

Purposeful Attestation

The Hadoop code could be altered to include a built-in attestation engine. This engine would require all Datanodes to attest their status (using TPM PCRs and signing keys) to the Namenode prior to being allowed to join the cluster, as well as on a periodic basis.

4.2.2 Experimental Trusted Computing Software Integration Design for Block Encryption and Key Protection

Looking at the feasibility and usefulness of the various trusted computing software integration concepts, implementing a block encryption engine with TPM-rooted key protections was the obvious choice for an initial prototype design, due to the overall value and improvement in data integrity and confidentiality this offered. Trusted computing components at the IT level combined with block encryption and key protection yield a trusted storage architecture. Below, the design details of this prototype will be discussed, followed by some alternative approaches based on findings, as well as performance and security results.

Given that data processing speed is tightly coupled to cluster read performance, encryption of any kind raises concerns. However, with recent results from Intel, claiming that AES encryption/decryption overhead can be reduced by using AES-NI, it made sense to use this in the design, attempt to verify these claims, and determine how much overhead one would incur in the circumstances, compared to no encryption [48]. In this

way, the owners of the HDFS data can make a reasonable trade-off decision regarding the security of their data versus their need for processing speed. Knowing the relative reduction in speed would be useful here.

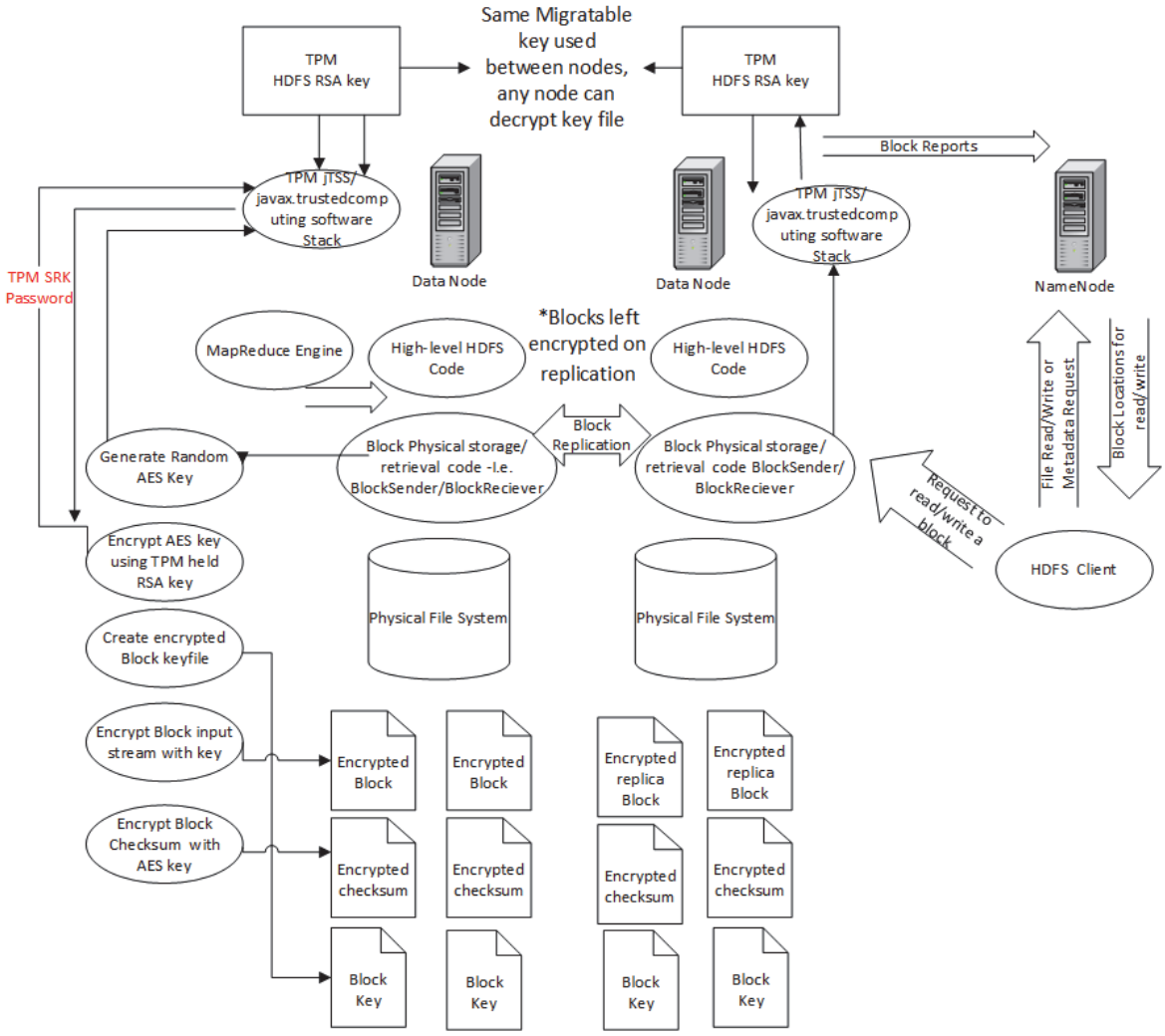

Figure 14. Architecture of Block-Based HDFS Encryption

4.2.3 Design Description

The above diagram illustrates the changes that were made to the HDFS codebase to implement a per-HDFS block AES encryption scheme with keys both generated and protected by the TPM. The version of Hadoop that was modified for testing was Cloudera

CDH4 distribution hadoop-2.0.0-cdh4.1.2. In the initial design, the desire, given the complexity of the code, was to limit cascading code changes. Although the HDFS code is relatively straightforward, the items that would potentially need to be changed to implement a protocol change would quickly grow. In this solution, each data block is individually encrypted and decrypted as the request comes in to store or read the block. Each node needs to have a shared RSA key that will be used to bind/unbind the AES key used for symmetric encryption. This was accomplished by defining the key as migratable when it was created, setting a migration secret, and importing the key into each node in the cluster. The key also needs to be created with a usage secret to prevent anyone with local access from using TPM tools to unbind data, and this secret must traverse to the node in a secure fashion. The primary code points for modification are the Datanode's BlockSender and BlockReceiver. The general process is that as a block arrives from a client, a random AES key is generated, and BlockReceiver wraps the FileOutputStream used to store the file with a CipherOutputStream implementing an AES/CBC/PKCS5Padding cipher. When the cipher was initialized, we used a random initialization vector that we saved as part of the AES key file. When the block was finalized, a key file corresponding to the block name (like the .meta file) was written to disk. Prior to that writing, the IV was combined with the random key and bound to the TPM RSA key. This way, the key and IV are always encrypted while on disk or in transit. When a client wishes to read a block (or when it is replicated to another node), BlockSender reads the file and passes the data and CRC values into a SocketOutputStream. When this is initiated, the code first determines if the request is for replication, and if so, the file is left alone. If it is being read by a client, we unbind the

AES key file, parse the IV, and create a new cipher and CipherInputStream to wrap the FileInputStream. Currently, this implementation encrypts all incoming data. To make this user selectable, a slight modification of the protocol could be made to flag the file to be encrypted.

As described in Intel's publication regarding AES-NI performance with Java, the JVM was configured to use Mozilla Network Security Services' (NSS) libraries to provide a native support layer for using the AES-NI instruction. This configuration is fairly straightforward, involving compiling NSS on the host platform and configuring the JRE to use it as the preferred crypto security provider via configuration files [49]. At the time of writing, the official JVM does not include hardware AES-NI support.

4.2.4 Software Modifications

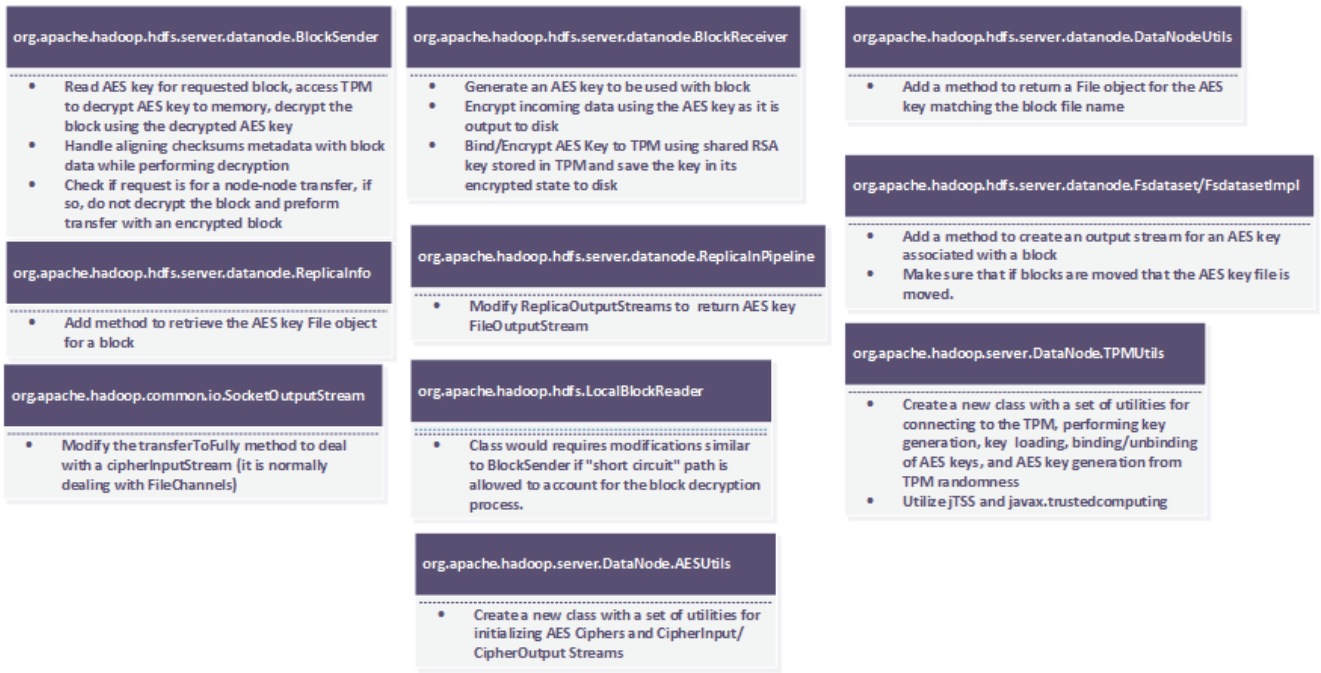

Figure 15. Simplified Summary of Modifications to HDFS Codebase to Support TPM-Protected Block Encryption

The above figure illustrates some of the code changes required to implement this form of encryption. The heart of the changes was the implementation of a TPMUtils and AESUtils class and modification of the Datanode's BlockSender and BlockReceiver classes. In addition, changes needed to be made to account for the association of a block key file. One difficult problem to address was the CRC checksums stored in the .meta file associated with each block. This file is used to validate that the incoming data matches the CRCs computed by the HDFS client for each chunk of the file. As the file is sent, it is split into 512-byte chunks and a corresponding CRC is created. Essentially, one needs to make this verification as the data comes in and save it as usual in the .meta file so that the decrypted file can still be verified. However, when we prepare to send the file back, we need to synchronize the decrypted data with the correct CRC from the .meta file to compare. Also, now that the data is stored encrypted, the periodic block scanner that detects corrupted blocks via comparison with the .meta file will fail. To correct this, we needed to add another .meta file that would correspond with the correct CRCs for the encrypted data. Otherwise, we would need to decrypt the file to validate that it is not corrupt, which is not ideal from a performance-overhead perspective. It is also possible that this scheme would cause issues with sync markers used for random access in sequence files, but this concern has not been addressed yet. BlockReceiver already has logic to check if the incoming block is from a Datanode (replication). If this is the case, we ignore any validation of the .meta checksum file, because we do not want to have to decrypt the block to validate this. Also, we leave the file encrypted for replication, and include the key file as part of replication. Another issue with this approach is in BlockSender, and the corresponding SocketOutputStream, as the SocketOutputStream

makes use of the FileChannel for a FileInputStream containing the block for OS-level optimizations. Likewise, BlockReceiver makes use of FileChannels for the data that is being written to disk to perform cache optimizations. Wrapping these constructs in cipher streams means that the FileChannel might not work as expected. The local “short-circuit” read/write path for an HDFS client located on the same node as the block has also not yet been addressed. This will have to be addressed, as this path is used to improve throughput, since the overhead of network sockets is not needed. The Hadoop configuration gives the option to disable the short-circuit path, which was disabled for testing. In a complete implementation, org.apache.hadoop.hdfs.LocalBlockReader will need to be modified to handle decryption and key retrieval for the short-circuit path.

The following chapter will explore the security implications and testing of the design.

SECURITY IMPROVEMENT RESULTS FROM EXPERIMENTAL IMPLEMENTATION OF TRUSTED HADOOP SOFTWARE COMPONENTS, VULNERABILITIES, AND PROTECTION OF SECRETS

The following table shows a summary of threats that were mitigated through the implementation of TCG components and the TPM-protected encryption scheme.

Security Risk	***Mitigation***
Data-At-Rest Protections	Implement AES-NI accelerated encryption of HDFS blocks. TPM allows for key protection of key materials
Key Protections For Encryption	Strong protection of RSA private keys via hardware TPM and binding of AES keys used for block encryption to private key. Protection of replicated keys via strongly protected key migration.
Decryption Of Exfiltrated Encrypted Data By Compromise Of Key Protections	AES keys bound to TPM and can only be decrypted by the TPM-protected key.
Data-In-Transit Protections	Built-in SASL can be augmented by leaving blocks encrypted during replication between nodes or by pre-encryption of data by client
Software Modification To Aid In Data Exfiltration	Boiler-plate TCG components (Trusted Boot, IMA/EVM, OpenPST) to ensure software is in known state via cryptographic signatures, PCRs, and srong attestation.
Compromise Of OS Running Hadoop, Enabling Data Exfiltration	Boiler-plate TCG components (Trusted Boot, IMA/EVM, OpenPST) to ensure software is in known state via cryptographic signatures, PCRs, and strong attestation. IMA PCR to detect unusual activity or changes in TCB.

Table 2. Summary of Security Improvements

5.1 Method

Formal methods of testing the relative improvement of security, or security of a system in general, can leave room for conjecture. For instance, we can complete white box, black box, fuzz, penetration, code analysis, vulnerability scanning, etc. but we really can't be sure that a system of this complexity is "secure". With that in mind, the purpose of the security testing in this research is not to attempt to prove that the proposed changes to Hadoop make it "secure" or "unhackable". The idea is more to go back to the trust

value judgment. What can we now prove about the system that we could not show before based on the vulnerabilities we sought to address? The above type summarizes the risk that were addressed and the specific mitigations. The method of testing I decided on goes to the worst case which is that an attacker has gained root access and/or can read network traffic. By taking this approach, I don't need to demonstrate security across a cluster, or worry about the higher level account control mechanisms and such. Compromise of one node means that the same results would be achievable on all nodes. So, for this test, we are assuming root access and describing what we could do before the proposed changes and why we "can't" (can't being a strong word in this context) or why it would be difficult to do the same things after the proposed changes.

5.2 Validation of Protections

In section 4.1.3, I explain the experimental set-up for the Node-Level Trust enabling components and how they help prevent or detect attacks. The general method for validation of the infrastructure security components and the software modifications are as follows. A Hadoop node was setup as a DataNode and NameNode. A second DataNode was setup to show replication protections. As stated, we assume root access has been gained. The first protection to test is compromise of the OS running Hadoop to aid in data exfiltration. In this test, we already have root access because we gained the password. So, we can do several things, of which will be detected by the Trusted Computing components. We could alter any key OS file, kernel, boot loader, firmware, or Hadoop packages. The protections that we have really do nothing to prevent this, as a root user may need to do these things at some point. What does happen is detection by two methods. The first is the IMA PCR. A production system should have a consistent

value in this PCR unless an administrator has done something with the system or if it has been compromised. The next protection comes at a reboot. The TrustedGRUB implementation will note any changes to the kernel or files in the checklist. Since this node is a NameNode, I've encrypted the metadata directory using dmcrypt that is bound to a TPM key and sealed to the known PCR state. Now, the system will fail to unlock this partition until action is taken to correct the failure. The detection feature alone does not by anything unless we check this, which is why a framework is needed to tie this data in to a cluster management system or monitoring of OpenPTS. The same mechanism detects tampering with the Hadoop JAR files, which an attacker might do to introduce a weakness to aid in exfiltration. The next item to check is network compromise, as in an attacker has gained access to a switch and can read raw packets. Since we are not encrypting blocks, the data on the wire between two nodes can easily be seen as encrypted using wireshark, even though the built-in SASL based network encryption was not enabled.

Lastly, we need to validate the protection of an attacker that has compromised the system without detection (somehow), and is now trying to exfiltrate data directly from the DataNode directories.

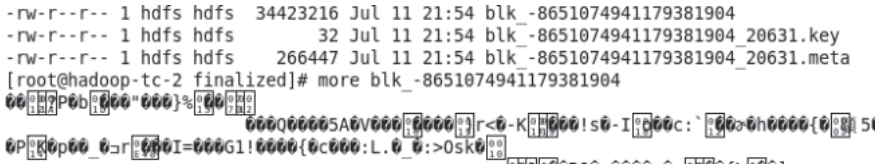

Figure 16. Protected Block

The sample HDFS Datanode directory output shown in the above image demonstrates the key advantage of the addition of block encryption from a security standpoint. Assume

this node has been compromised by attackers wishing either to exfiltrate or modify the data stored in the HDFS. In a traditional configuration, they could simply traverse to the configured directory for the distributed file system and begin looking at or modifying the block files (modifying would cause a checksum error, so they would have to modify the binary meta file as well, and all replicas of this block). This would be the most typical, straight-forward attack on a root-compromised system. As text data is a typical format for Hadoop, these files could be read without any further work. Now that the file has been encrypted at the block level, the attackers can no longer use the block files as-is. They will have to do one of two things. They will either have to convince Hadoop to decrypt the file for them by compromising the access control to run a MapReduce job on the dataset or to gain access to the Hadoop application via a compromised account, and ask Hadoop to copy the file out of HDFS (at which point it will be decrypted); or they will have to intercept the raw AES key while it is in use. Although the AES key is never unencrypted on disk, it is in memory while files are being encrypted/decrypted. Using a new key for each block has the advantage of limiting the time each key is in memory. Also, modern OSs have protections against simply dumping the memory of a live system, although more advanced attacks on memory are possible. Lastly, the attackers could convince the TPM to decrypt the key file for them. To do this, they would need to acquire the usage secret for the key. The feasibility of this attack depends on how the secret gets to the TPM; as it is only needed to load the binding key, it needs to be in memory for only a short time. Having an administrator manually enter the usage password on system initialization is an option, but besides being impractical for a large cluster, it would be potentially vulnerable to a key-logging attack (although the trusted infrastructure design

should prevent this). The initial architecture does not address the issue of secure input of this secret; however, a mechanism for releasing this secret that should mitigate the risk of its compromise will be described. Of these attacks, the attack against a user account with access to these files is probably most likely, or manipulation of the weak token model as described in the threat model to convince the Hadoop software to allow access and convince the system to decrypt the file.

Using a TPM RSA key as a binding key to protect AES keys used for file encryption/decryption has the advantage of creating a situation in which the file can only be decrypted on a system that has this RSA key installed in the TPM. This gives a certain advantage over the use of a software-based key container (such as the Java Keystore that is being used in the Intel HDFS encryption scheme), because those keystores are not tied to a platform, and the keystore has to be opened at some point for the application to be able to use the private key. This action would typically be susceptible to an attack in cases in which the attackers have acquired the password to unlock the keystore. If they are able to do this via a keylogger or social engineering attack, they could exfiltrate the data and unlock the keystore, decrypt the AES keys, and use them to decrypt the data. They could also do this on the Datanodes directly, but it is possible that an admin might detect this activity more quickly than a more subtle exfiltration via covert channels. By using the TPM, however, I significantly limit the effectiveness of this type of attack by virtue of the fact that the private key is stuck in the TPMs of the participants in the cluster; hence, the AES key can only be decrypted there. To validate this, I tried to decrypt a file using the AES key file manually with OpenSSL. Since the AES key file is bound to the TPM, OpenSSL fails. The one exception here is the fact that a migratable

key was used among all nodes in the cluster so that data would not need to be decrypted during replication and encrypted again to another key. If attackers gained access to the migration secret via an attack against the admin, or if the admin became a malicious insider, this security would be broken. As mentioned, we need to make it difficult for attackers to get the secret that prevents the binding key from being used. The figure below demonstrates a way to transmit the usage secret to participants in the cluster after their integrity has been verified. Upon initialization of the HDFS code, the Datanode checks in with the Namenode. Typically, the first thing that it will do is send a block report to the Namenode; however, in this scenario, the Namenode will respond with a nonce. The Datanode then uses a TPM signing key designated for this purpose to return a quote of the PCRs for the system that is signed with the private key, and a signed hash combing the nonce. To enable this without a TPM key secret, we have to use the TPM WELL_KNOWN_SECRET as the usage key for the storage root key and the signing key. The Namenode has the public key for the Datanode signing key, which it uses to validate the returned quote and compare it against a known value. If this comparison shows that the system is in a known state, the secret is returned as a hash value after being encrypted to the public-key part of the signing key. Now, only the Datanode's private key can decrypt it, which it does, immediately using it to load the binding key into memory and overwrite the memory used to hold the secret. The Namenode was used as the attestation server, but this could be another device if desired. This design places extra security requirements on the Namenode, since its compromise would break this mechanism. Also, an admin is shown entering the secret into the Namenode, at which point it only exists in memory as an encrypted blob. The use of a late-launch technology called Flicker, based

on Intel TXT, was considered to provide a hardware-isolated environment for input of the secret as an area for future research.

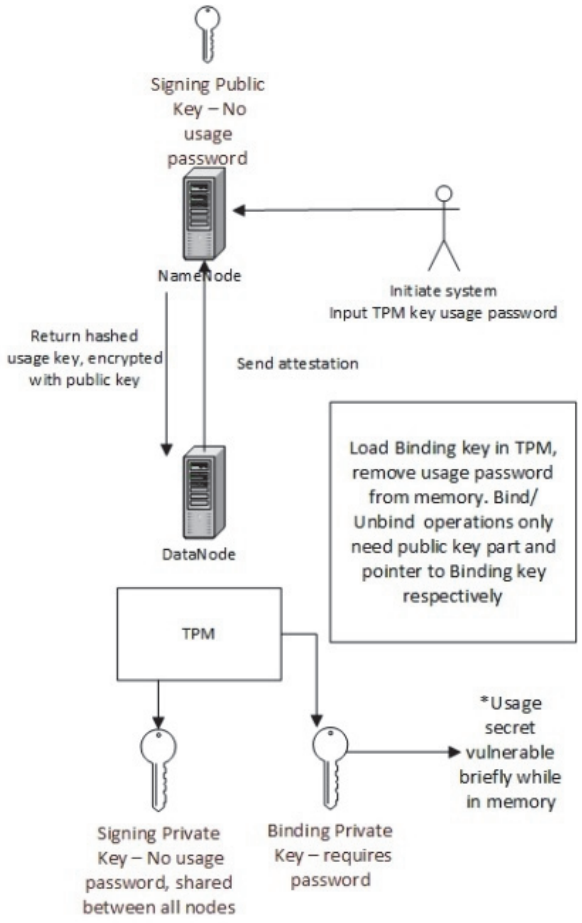

Figure 17. Binding Key Secret Protection Mechanism

In the following chapter, a discussion of the performance implications of implementing the AES based encryption with TPM rooted key protections will be presented.

PERFORMANCE RESULTS OF EXPERIMENTAL TRUSTED HADOOP HDFS STORAGE PLATFORM

To determine the performance implications of the prototype encryption design using AES-NI for encryption acceleration and the TPM for key protection, a benchmarking suite to simulate the read/write operations of HDFS was created. Most tests were conducted with a 128MB random text file, representing a typical HDFS block size. The text of a file of over 5GB was also used as a test to compare the stock Cloudera CDH4 distribution hadoop-2.0.0-cdh4.1.2 HDFS with the modified version. For all tests, 128-bit AES keys were used, which the US government approves for use up to "Secret"-level classification. Note that all tests were on a single node. Future work should test scaling on multiple nodes on more enterprise-class hardware. Results are shown as a random sample of five runs of the test suite.

The test system specs are as follows:

- HP 8260p Core i5 2520M w/ vPro 2.50 GHz 6GB RAM two SATA 3.0GBps HDDs
- Infineon TPM 1.2
- Native OS Disk Speed Results: Cached Reeds 4465 MB/s Buffered Disk Reads 116 MB/s 128 MB writes w/sync 78.57 MB/s
- OS: CentOs v 6.3, Oracle JRE 1.7_25

	TPM AES Key Gen	Java key Gen
r1	633	73
r2	474	73
r3	463	74
r4	471	79
r5	481	79
Avg	**504.4**	**75.6**

Table 3. TPM vs Java SecureRandom Generation of 128bit Random AES keys in ms.

The table above shows a comparison of time to create a random 128-bit AES key using the javax.trustedcomputing API call to the TPM randomness generator versus the Java SecureRandom mechanism of random key generation. Since we only need to generate keys while files are being written, this delay may not be significant in terms of performance; however, we do begin to see the cost of using a TPM command in this simple test. It is likely that using the TPM for AES key generation is not buying much value in terms of security compared simply to using the software generation.

	Encryption from Disk AES-NI	Encryption from Disk NO AES-NI	Actual Encryption Time No AES-NI	Actual Encryption Time AES-NI	Encryption from RAM AES-NI	Reference Java Copy Operation	Reference Disk Write w/ sync
r1	369	1857	1245	94	378	263	1666
r2	412	1626	1212	60	186	260	1548
r3	407	1650	1227	61	186	270	1650
r4	471	1695	1217	61	234	265	1677
r5	424	1656	1220	62	182	263	1604
Avg ms	416.6	1648	1224.2	67.6	233.2	264.2	1629
MB/s	307.24916	75.436115	104.55808	1893.491	548.885	484.4815	78.5758134
	74.75% Faster			94.478% Faster	44% faster		

Table 4. Comparison of Encryption Time in Milliseconds and Throughput for 128MB Text File Equivalent to One Block

The above table shows the results for encryption of a 128MB text file. For the encryption from disk with AES-NI test, the file was read in from disk as a FileInputStream and written out to a FileOutputStream wrapped with a

CipherOutputStream using the 128-bit AES cipher. The results indicate the effects of OS disk buffering, as the total throughput was faster than the possible disk IO throughput, as shown in the far-right column. Throughput was about 57.5 percent slower than a simple copy operation, although this test is likely skewed by the effects of caching. Comparing this test with the same method but using the built-in Java crypto provider without AES-NI acceleration, a significant decrease in throughput is observed, with the AES-NI support presenting a 74.75 percent increase in throughput, or about a 4x increase. The next test aimed to determine the time spent on the actual encryption operation. To do this, the file was fully read into memory and encrypted in memory, with only the encryption operation measured. Again, a significant increase in throughput of 94.5 percent (or 18x) is observed using AES-NI versus the standard library. More importantly, the actual cost of encryption appears to be adding about 16.2 percent in overhead, not counting the key generation or key protection stages. Lastly, the encryption-from-RAM test showed the time taken to encrypt data already in memory and to write to the disk.

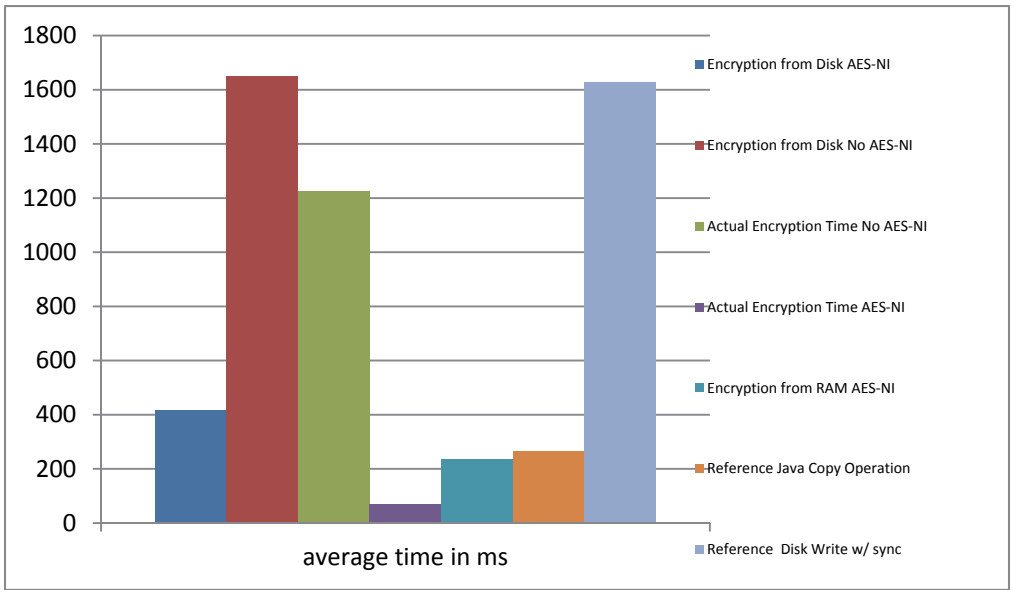

Figure 18. Comparison of Encryption Times in Milliseconds

	Decryption from Disk AES-NI	Decryption from Disk No AES-NI	Actual Decryption Time No AES-NI	Actual Decryption time AES NI	Decryption from RAM AES-NI	Read Throughput without decryption
Run 1	1633	2794	1525	194	378	1439
Run 2	1614	2730	1298	186	338	1425
Run 3	1648	2743	2542	194	347	1454
Run 4	1680	2671	1504	161	310	1519
Run 5	1700	2683	1544	208	362	1492
Avg ms	1655	2724.2	1682.6	188.6	347	1465.8
MB/s	77.34138973	46.9862712	76.07274456	678.685048	368.8760807	87.32432801
	12% slower than raw 63.83% faster than No AES-NI	37% slower than AES-NI		88.79% faster than no AES-NI		

Table 5. Comparison of Decryption Time in Milliseconds and Throughput for 128MB Text File Equivalent to One Block

The above table shows the results of decryption of a 128MB text file. For the decryption from disk tests, the file was read in from disk via a FileInputStream wrapped in a CipherInputStream using the 128-bit AES cipher and written to a new file in an unencrypted state. The far-right column shows raw-read throughput for the test system as a comparison. The results show that decryption speed was 63.83 percent faster using AES-NI than the standard library, representing a 1.6x increase for this test. Next, the actual time spent decrypting the file was isolated by first loading it into memory and decrypting it to memory. This test indicated an 88.79 percent increase using AES-NI (9x faster), and that the decryption process was adding about 11.4 percent in overhead.

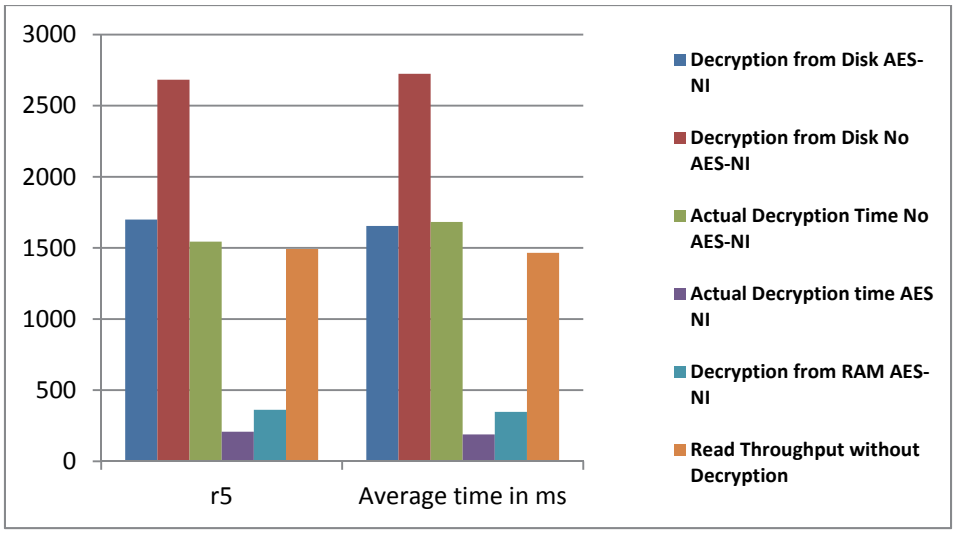

Figure 19. Comparison of Decryption Times in Milliseconds

	TPM AES Key Generation 128 bit	Create Binding Key	Load Binding Key	Bind/Store AES key with TPM	Unbind AES key and Decryption of file AES-NI	Unbind AES key and Decryption of file NO AES-NI
Run 1	633	19914	1516	69	2041	3276
Run 2	474	29095	1512	66	5059	3310
Run 3	463	21714	1507	74	1834	3144
Run 4	471	11995	1509	122	1861	3395
Run 5	481	11737	1514	106	1678	3427
Avg ms	504.4	18891	1511.6	87.4	2494.6	3310.4
MB/S					51.3108314	38.66602223
					40.6% Overhead vs AES alone	

Table 6. Performance of Operations Using the TPM

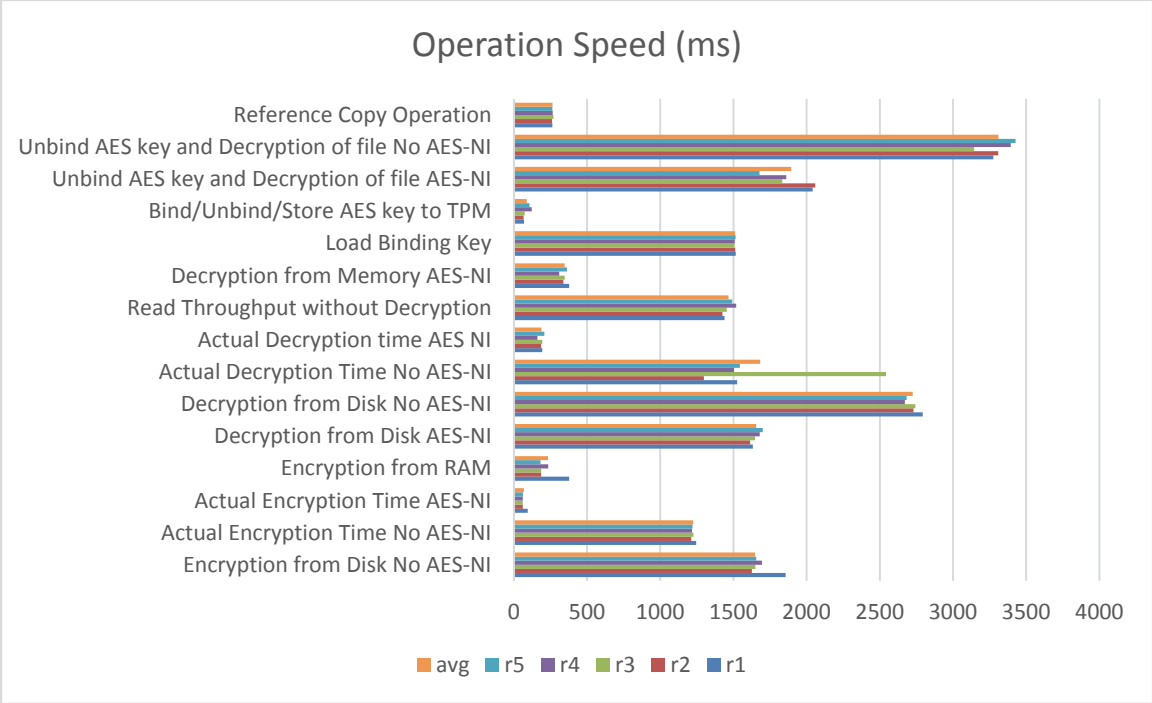

Figure 20. Operation Speeds in milliseconds

The above table and chart show the speed results of using the TPM for functions required in the design. The first column shows the time required to generate a random AES key using the TPM, with an average of 504ms. Interestingly, this process is longer than the actual encryption of a file, which took 416ms. The second column indicates an average of almost 19 seconds to generate a new 2048-bit RSA key. However, this operation would only need to be done once, on initial setup of the node, and does not affect functional performance. Likewise, loading the binding key will only happen when the node comes online, and will not affect the subsequent operations. Binding the AES key to the TPM's RSA binding key and writing the file out averaged 87.4ms. Since the initial architecture does this once per block, the AES key generation is a combined 504ms for key generation and 578ms of overhead for a write operation, increasing the time required to encrypt a 128MB file by 139 percent. This overhead would decrease relative

to the size of the file, becoming less-significant the larger the file. Also, the overhead can be reduced to 39 percent by using the SecureRandom Java construct to create the AES key instead of using the TPM, thus adding only 163ms to the write operation for a 128MB block. Next, the decryption process was tested by first unbinding the AES key using the TPM and decrypting the file with that key. Here, an average overhead of 40.6 percent was observed versus the AES decryption process alone with a plain AES key. The TPM unbind operation added about 840ms to the decryption process on average. Again, the percentage of overhead would become less-significant as the size of the block file increased. For instance, for 256MB blocks, the overhead would be reduced at least 20 percent; for 512MB blocks, 10 percent. It is not uncommon to configure an HDFS cluster with larger block sizes to tune for very large files.

	TPM unbind AES key and decryption of file AES-NI	Decryption from disk AES-NI	Read throughput without decryption
Avg Time (ms)	2494.6	1655	1465.8
MB/s	51.31	77.34	87.32
	33.657 % overhead vs. AES alone	12% slower, raw	
	41.37% slower than raw	63.83% faster	

Table 7. Comparison of IO Throughput

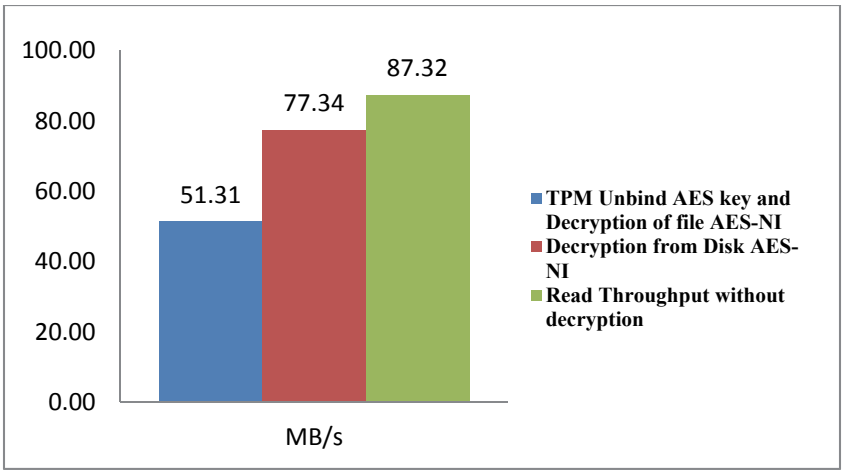

Figure 21. Comparison of Throughput

The two tables below shows comparison of performance on 128MB and 256MB file sizes. Note that these results are based on the micro benchmark suite, and results on actual Hadoop data will likely be a bit different due to the nature of how data flows in and out of Hadoop (i.e. added overhead of network and the fact that we are encrypting as the block comes in one chunk at a time instead of encrypting a stored file as shown on this test). However, this test does isolate the overhead induced by the TPM key management piece and its relative effect on overall IO between 128MB and 256MB blocks, showing the reduction in significance of these operations as the block size increases (since the TPM times are the same regardless of block size). These results also show the results of file system caching on performance, which is something that is difficult to isolate on a production system. By forcing a disk sync instead of allowing the encryption process to complete on a buffered file, the results show 256 MB files encrypted with TPM key protection (SecureRandom key generation) come within less than one MB/s of the raw system IO (76.7 and 77.2 MB/s). 128 MB files, on the other hand, reduce in speed to 55.4 MB/s, showing the greater overhead significance of the

TPM operations when dealing with a smaller block size. This again shows that disk IO can become more of a bottle neck than the actual encryption and key management process. This is evident in the results using buffered output. On a 256 MB file, 232 MB/s can be achieved, even with the encryption and TPM key protections. So, in terms of scaling of the results to other platforms and more advanced hardware, the following would remain true. First, TPM operation speed will be consistent on any platform using an Infineon 1.2 TPM. This is the most common TPM in use in enterprise class hardware, and the bottleneck comes from the LPC bus and the inexpensive nature of the design. AES-NI performance will be relative to the clock speed and buffering capability of the platform, so this will likely vary from machine to machine, however, the relative percentage in speed difference would likely be similar. Lastly, unless the system can achieve greater than 232 MB/s Disk IO, the encryption and key management design will not slow down the IO speed. Interestingly, decryption time is consistent for both block sizes at around 51 MB/s. This could be due to the fact that the file must be read into the cypherOutputStream from a fileInputStream and as a result, OS buffering does not help in decryption.

	Encryption with TPM Key Generation/Protection 256MB with Buffered output	Encryption with TPM Key Generation/Protection 128MB with Buffered Output	Encryption with TPM Key Generation/Protection 256MB forced sync	Encryption with TPM Key Generation/Protection 128MB forced sync	Encryption with TPM Key Protection 256MB forced sync (SecureRandom keygen)	Encryption with TPM Key Protection 128MB forced sync (SecureRandom keygen)	Decryption with TPM Key Protection 256 MB	Decryption with TPM Key Protection 128MB	Copy with Sync (reference) 256MB	Copy operation (reference) Buffered 256 MB
ms	1103	1007	3338	2310	3441	1880	4980	2494.6	3317	367
MB/s	232	127	76.7	55.4	74.4	68.08	51.41	51.23	77.2	697.54

Table 8. Comparison of 128 and 256 MB File Operations

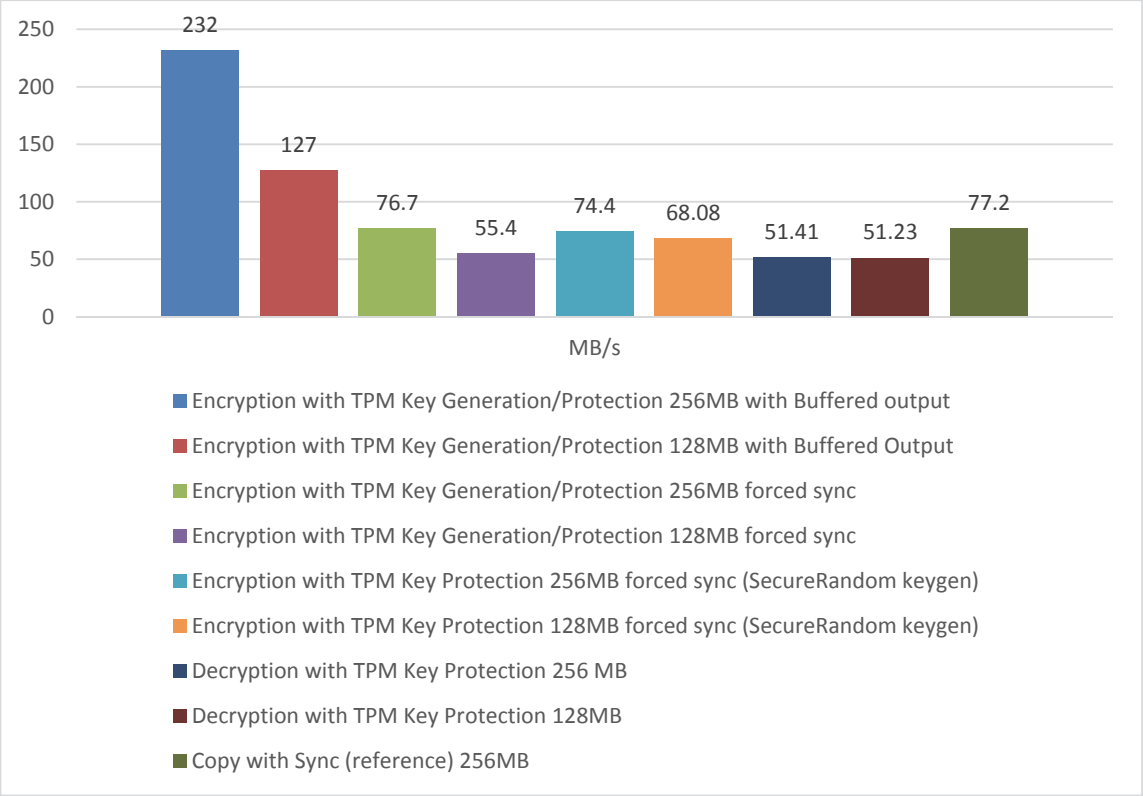

Figure 22. Throughput Comparison of 128/256 MB Blocks

	Encryption Disk to Disk AES-NI	Encryption Disk to Disk ***no*** AES-NI
Time (ms)	142669	170175
Operation MB/s	41.09	34.45
Total MB/s	81.61	68.9
	18.4% Faster	

Table 9. Encryption of a 5863MB File

In the above table, the results of encrypting a large file with and without AES-NI are shown, including reading the file from disk in its encrypted state and writing the file out decrypted. The effects of a disk IO bottleneck are likely being observed in this result. However, this demonstrates that for large file operations, AES-NI can produce near-

native IO performance as it begins to keep pace with the disk IO. The disk has an 82.95MB/s average raw-read/write, showing only a 1.2 percent decrease in IO while using AES-NI. A faster disk subsystem would likely show a more significant difference between AES-NI and the standard library.

Other tuning parameters to consider are the input/output stream buffer sizes, as they should be sufficiently large to keep pace with the IO subsystem speed. The testing confirmed the general recommendation of a 4k or 8k buffer. No appreciable difference was noticed at higher buffer sizes; however, significant decreases in performance were seen when using smaller sizes.

Interestingly, the addition of the AES encryption scheme made little difference in speed when applied to the import of a large file from one local hard drive on the test Datanode into HDFS (residing on another internal hard disk). This again demonstrates that the Java IO throughput on this particular system was more of a bottleneck than the hardware-accelerated AES encryption. Also of note is that CPU utilization was within one percent for both runs, demonstrating the advantage of the processor instructions.

Results:

- 5863MB file with stock HDFS: 2:43.28, 22% CPU
- 5863MB with AES encrypted HDFS: 2:49.38 21% CPU

Summary of results for 128MB block file:

- Encryption throughput read/write – 74.75% faster with AES-NI - 4x faster
- Raw encryption throughput - 94.5% faster using AES-NI - 18x faster

- Encryption overhead introduced using AES-NI – 16.2%
- Decryption throughput read/write – 63.83% faster with AES-NI -1.6x faster
- Raw decryption throughput – 88.79% faster using AES-NI - 9x faster
- Decryption overhead introduced using AES-N1 11.4%
- TPM create AES key, encrypt file, bind AES key 139% overhead vs. AES alone, reducible to 39% by using SecureRandom instead of TPM.
- TPM unbind of AES key and decryption 40.6% overhead vs. AES alone

Intel claims in the Hadoop Crypto Design [50] a 5.3x improvement in encryption and 19.8x improvement in decryption using AES-NI; although my results do not support this level of increase, it is possible that it is due to differences in system configuration, particularly in terms of IO throughput and input sizes. The results are in line with the microbenchmarks of the AES encryption process that Intel published, but given the differences in system configurations, it is hard to make an exact comparison. However, it is clear that the use of a crypto library supporting these hardware instructions will make a significant improvement to the encryption overhead burden.

The proposed encryption engine affects storage retrieval/storage speeds relative to the base system. The percentage of overhead related to the TPM and encryption operations noted was related to raw HDFS performance. Algorithm performance degradation due to this overhead will be tied to the number of stored blocks that the algorithm will process, and the size of these blocks. The perceived degradation of the hardware-accelerated encryption overhead may not be perceivable due to IO subsystem bottlenecks being more significant than the CPU time taken to perform the encryption. However, TPM operations in the initial model will become significant if a large number

of blocks is being read or written, due to a minimum of one TPM operation per read/write request to handle the AES key binding or unbinding (which is likely dealing with a typical large Hadoop workload). In the following section, a discussion of some alternative designs that will reduce this overhead will be presented. In the current model, however, using large block sizes will reduce the significance of the key retrieval delay. Considering core data-warehouse style workloads, a typical task will only read the data files once in the map tasks, creating an output file. The output files are read again in the reduce tasks, and then there is the creation of a final output file. In this case, the reduction of algorithm performance is directly correlated to the overhead of the AES key bind/unbind and encryption of each block read from the input, the number of blocks of intermediate output files created, and the final output file. However, it is also possible to implement interactive algorithms by cycling through multiple map tasks. In this paradigm, the more times input is read and intermediate files are written, the greater the cumulative overhead—particularly in the initial design that was implemented, where the AES key is bound/unbound using the TPM upon each access request for the block. So, for example, if each block read/write experienced a 87ms delay due to the key bind/unbinding process, the total delay would be delay = (87*(number of map-task blocks read)+ 87*(number of map-task blocks written)) + (87*(number of reduce-task blocks read)+87*(number of reduce-task blocks written)). This could quickly add up to a lot of overhead, particularly with an iterative algorithm in which total delay would be a summation of the delay for each iteration. For instance, 100 GB of data with a 128 MB block size would result in 8000 blocks. Processing each block once would result in an 11.6 minute delay from the AES key unbinding to read all the blocks. Presumably each

map task would also generate an output file which, again, would add delays at least equal to the number of tasks, and the reduce tasks would add delays equal to the number of map output blocks and the number of generated result blocks. Again, in a data warehousing application, where it would be acceptable to add 11.6+ minutes to a job that is not expected to produce real-time results, this may not be a significant factor. The following chart illustrates the overhead introduced by implementation with two common HDFS benchmarks, TestDFSIO and TeraGen/Terasort. These benchmarks are bundled in the Hadoop-examples.jar. The TestDFSIO benchmark is a read-and-write test for HDFS. It is helpful for tasks such as stress-testing HDFS, to discover performance bottlenecks [51]. It accomplishes this via running map tasks to generate random files of specified size and quantity. The goal of TeraGen/TeraSort is to generate and sort 1TB of data (or any other amount of data desired) as quickly as possible, combining a test of the HDFS and the MapReduce layers of a Hadoop cluster [51].

Metric	Base	Encryption w/ TPM protection of keys
Number of Read Ops	33	33
Number of write Ops	12	12
Time (ms) spent by Map Tasks	278724	281496
Time (ms) spent by Reduce Tasks	17919	18891
Total MB processed	5120	5120
Average IO MB/s	34.02	28.64

Table 10. Results of TestDFSIO on 10 512MB files with 128MB blocks

Metrics	Base	Encryption w/ TPM protection of keys
Number of Bytes Read	1000394182	1000394182
Number of Bytes Written	1000000000	1000000000
Read Operations	20	20
Write Operations	1	1
Map time (ms)	108164	109944
Reduce time (ms)	17054	18782
Total Time (sec)	54	57.4

Table 11. Results of Terasort on 954MB of Data with 128 MB blocks

Based on the presented results, several alternative designs were conceived to potentially improve the performance of the key management and protection aspects of the encryption design. The next chapter will discuss these designs concepts, as well as the latest proposed encryption framework that has been proposed for Hadoop Common.

DISCUSSION AND FUTURE WORK

7.1 Applications

Looking at the performance limiting factors of the initial design, the application set that would be appropriate to this design becomes a bit clearer. First, the sensitivity of the data being stored in the Hadoop cluster should be considered and alternative approaches to data security should be considered. One common method in industries like health care is anonymizing data via an appliance or pre-processing the data. In this way, no personally identifiable information is actually stored in the cluster. If actually sensitive data must be stored, then an evaluation of how likely the cluster will be attacked should be completed. For instance, how connected will it be to other systems, networks, people, and the internet? Also, what type of algorithms will be used and do they need to be near real time? If the purpose is more of a warehousing system where near real-time analytics is not as much of a concerned with mostly non-iterative algorithms, then the encryption and key protection scheme described could be used without issue. Use cases would include defense data collection systems or financial analysis systems.

7.2 Alternative Designs

During testing, some performance-limiting factors of the design (which were expected to be an issue) related to the performance of the TPM were confirmed. In the experimental design, a new AES key was created for each individual block, and these keys had to be bound/unbound as they were needed, using the TPM. Depending on the block size used, this may represent a significant cumulative overhead, not to mention the effects of concurrent read requests. In other words, for smaller blocks, the delay is more pronounced than for larger blocks, and concurrent requests would have to wait for TPM

availability. Using the default configuration of 128MB blocks, blocks can be decrypted in less time than the TPM unbind time. Although this can be mitigated by increasing the block size, it would still be advantageous to use the TPM less. As a result, the following design alternatives demonstrate ways in which the TPM can be used less-frequently, but still be used to secure the symmetric keys. In addition, given the likelihood of the crypto framework proposed in Hadoop-9331 becoming mainstream, we take a brief look at its design and the potential of integrating with it.

7.2.1 Key Pools

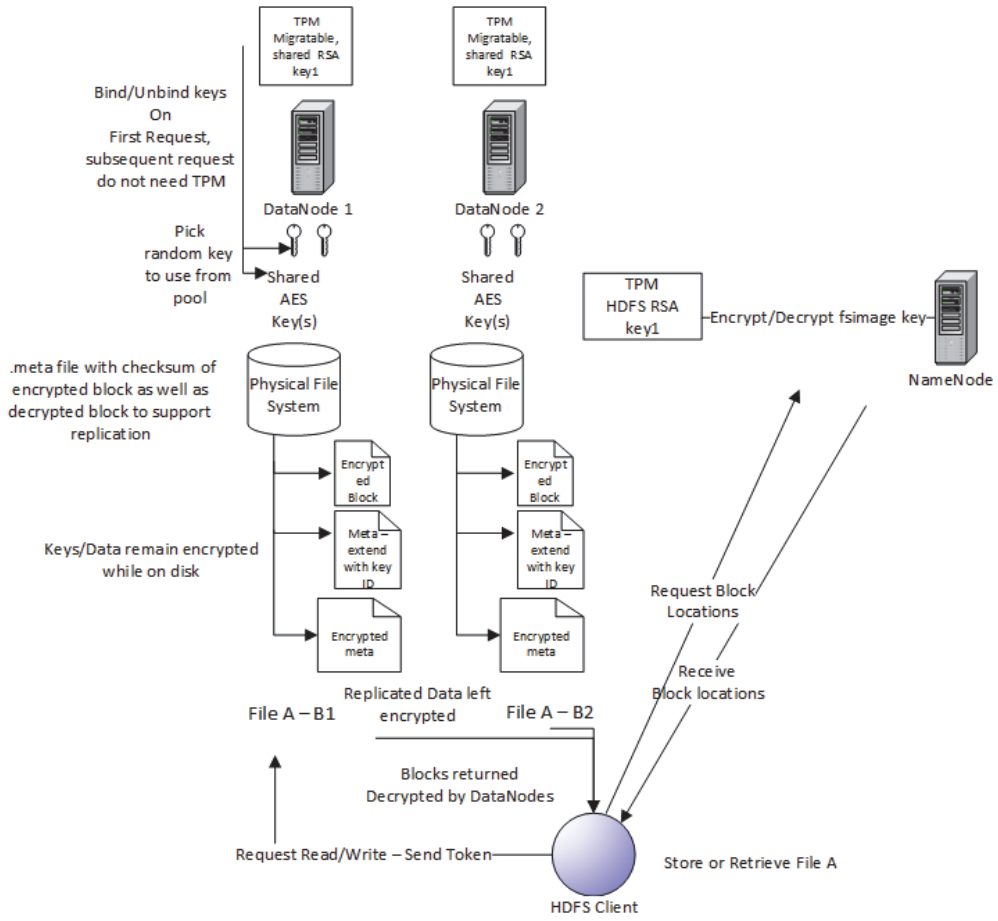

Figure 23. Alternative Design

In Figure 23, a method for file encryption similar to the original model is presented, but now designed to require far fewer bind/unbind TPM operations, hence bringing throughput back in line with encryption overhead only. To accomplish this, we treat the encrypted blocks in a way similar to how a traditional encrypted file system would work by using a limited number of AES keys. But instead of using just one AES key for all blocks, the design calls for a bank of AES keys to be shared among all Datanodes; perhaps one for each user account, or just some arbitrary configurable number. The idea is to limit the effects of a cascade compromise if one key is compromised. Each AES key is still bound to a TPM RSA key, and hence can only be decrypted on a system containing a TPM with the shared RSA key. When a block is created, the Datanode determines which AES key to use, and makes an entry in the metadata file indicating which key was used (if any). When the block is accessed, the Datanode checks the metadata first to determine if decryption is necessary and if so, what key to use. The Datanode would maintain a key broker object in memory that would only return a reference to the decrypted key to use in the block decryption process. Also of note in the design is the addition of an encrypted metadata file that would be used for block transfers among nodes for replication events to verify the integrity of the file as well as the periodic block scan that checks the integrity of the file. In this way, the traditional metadata file is only used by a client to verify that the decrypted value is correct. The design also shows the Namenode FSImage (and edit log) being encrypted with an AES key bound to the TPM. Since HDFS is not metadata-intensive, the overhead of this design would be negligible. The attack vectors on this design are similar to those of the original design. The simplest attack would be to compromise a user account and

use the HDFS client to output data in the clear. An attacker gaining root access could conduct a code replacement attack or potentially retrieve the AES keys from active memory, but if infrastructure enhancements have been put in place, a code replacement attack would be detected. Defending against a compromised account is trickier, and can really only be addressed by implementing an improved authentication scheme such as two-factor authentication requiring a hardware token, or TPM with an RSA key. An interesting possibility would be to require all participants in an HDFS infrastructure to have a TPM; the Datanode could generate an AES key and bind it to the remote participant's public key, hence limiting where the resulting data could be decrypted.

7.2.2 File-Oriented Encryption

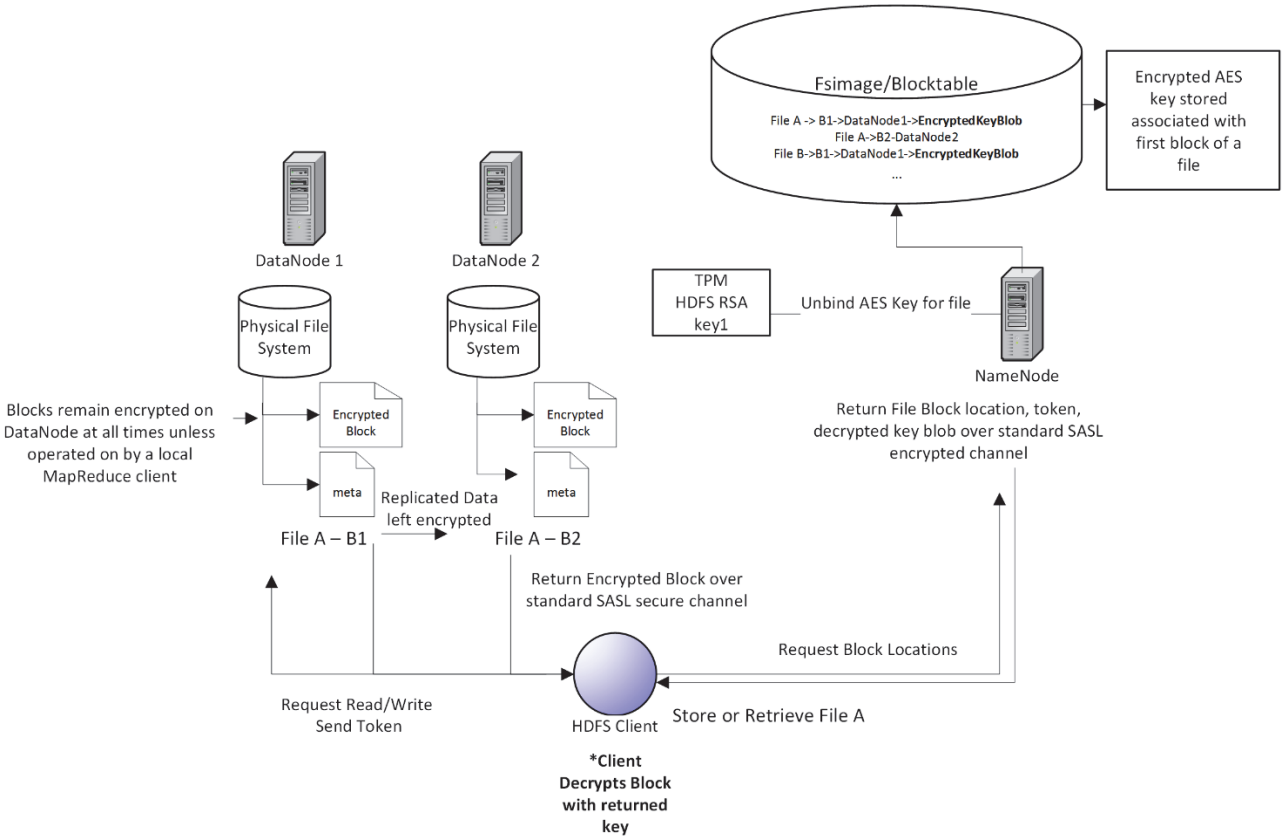

Figure 24. File-Oriented Encryption with Client Encryption

In this alternative approach, an architecture in which each logical file (as opposed to each block) has a single AES key is presented. This key is bound to a TPM RSA key on the Namenode. The Namenode's metadata would be extended to include an entry pointing to a file containing the AES key. In this scenario, the client is responsible for encryption/decryption, with the client possibly being on a Datanode as a MapReduce job. The reason for this is that the Datanodes currently have no knowledge of files; they simply store and return blocks. When the client wants to read or write a block, it first contacts the Namenode, as it currently does, except the returned information will include an AES key to use. It is essential that the SASL encryption be enabled in order for this communication to protect this key. Alternatively, if all clients were required to have a TPM, we could bind the AES key to a public key corresponding to a private key in the client's TPM, ensuring it can only be decrypted on the intended target. I envision that the client would then essentially encrypt the file that it is sending as a series of blocks. However, there is no reason why this key could not be used as part of the proposed HADOOP-9331 crypto scheme, using the compression filter. Also, it is possible that if we wanted to stick with the concept of allowing the Datanodes to handle encryption, we could do so by asking the Namenode to return the key associated with the block for each request involving the block.

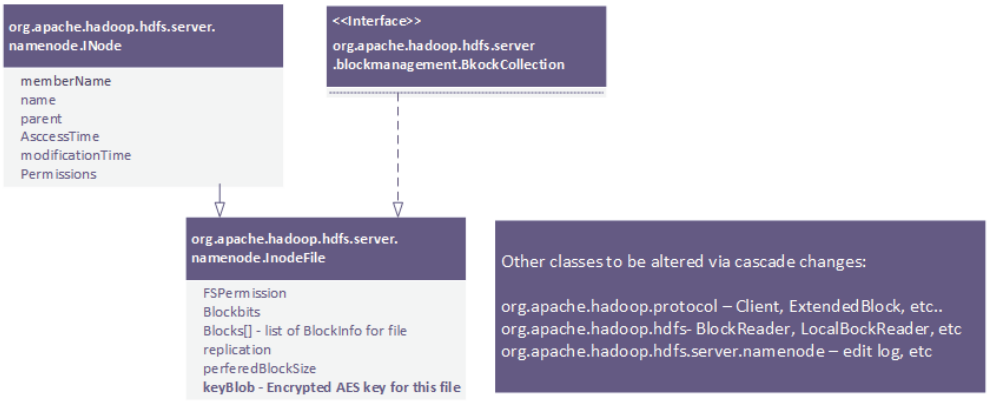

Figure 25. Simplified Changes to Namenode Inode Class to Include AES key and Cascade Changes for File-Oriented Encryption.

Since we only need to unseal a key once per file request, the delay from using the TPM is reduced to one operation per read or write. When dealing with large files, like those that Hadoop was designed to deal with (i.e., in the TB range), the time cost of an initial delay in unsealing an encryption key becomes less significant, and the overhead concerns return to the encryption/decryption process. Potential attacks on this architecture are similar to those on the other designs, including a code replacement attack or compromising a user account and asking HDFS to decrypt the file. In addition, since the encryption key is transmitted on the network, any compromise of network protocol would reveal the key.

7.2.3 HADOOP-9331 Crypto Framework

In 2013, there was much parallel work in the area of Hadoop security, specifically using AES-NI for encryption. Of particular note is the introduction of Intel's Hadoop distribution supporting AES encryption, and their push of their crypto framework back to Hadoop common as JIRA HADOOP-9331 [50]. Intel refers to their encryption implementation as "Project Rhino," explaining, "Encryption is transparent to users, can be applied on a file-by-file basis, and works in combination with external key

management applications. Java KeyStore is currently supported, and future versions will support a broader range of standards-based key management solutions" [7]. For instance, there are a number of products that manage keys via a hardware security manager and implement a key exchange protocol. The solution of using the TPM might be more practical for an organization that does not have access to one of these solutions, or does not want to connect that part of its infrastructure to its big data infrastructure. This research towards encryption integration began prior to publication of this proposal and the release of Project Rhino; however, given the corporate backing of this crypto framework, it is likely that it will become mainstream in Hadoop common in the future, and it would be prudent to evaluate how one could introduce additional protections into this scheme in a way similar to the presented designs. In addition to Intel's work in this area, a similar AES encryption scheme was proposed at GPC 2013, which also makes use of the idea of implementing AES encryption as a compression codec, and of hardware AES acceleration, yielding a seven percent computational overhead [27].

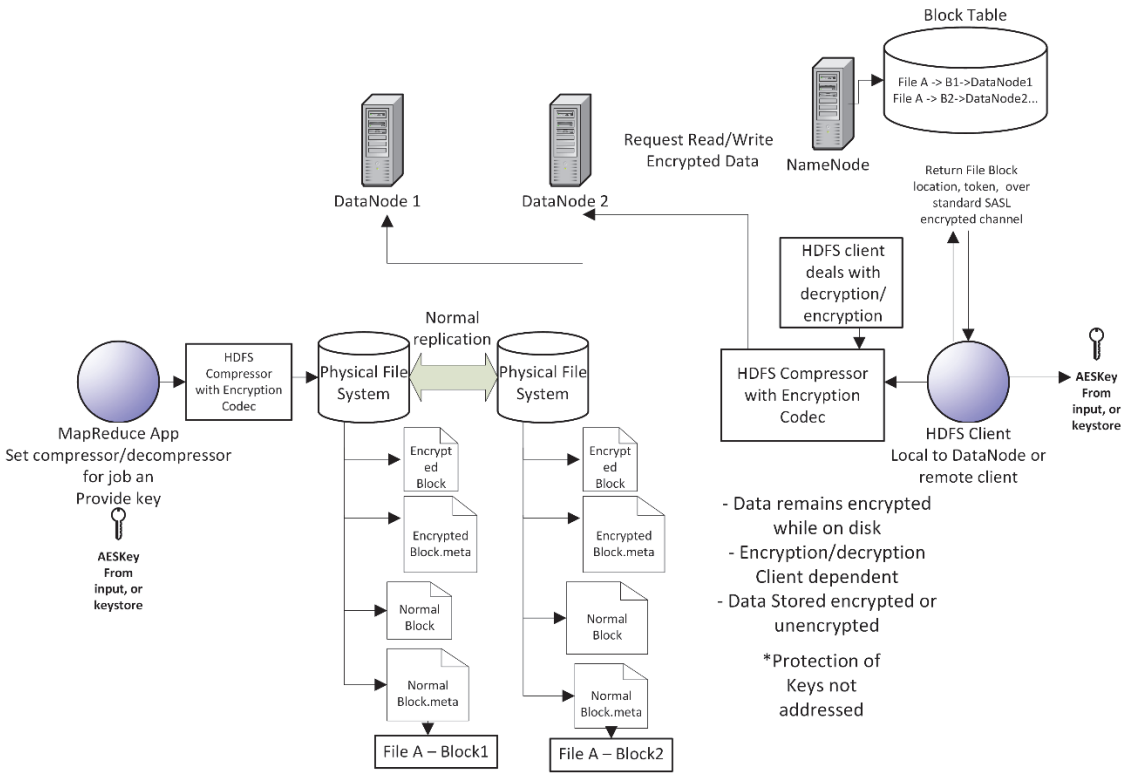

Figure 26. HADOOP-9331 Hadoop Crypto Codec Framework and Crypto Codec Implementations

The figure above gives a simplified look at how the crypto design is implemented. The key concept is that the existing data compression framework is used, with the addition of an AES codec, which provides some benefits over the approach taken in the experiment. Intel also made some extensions to MapReduce to handle keystores for AES codecs. The benefits of this design include a reduction of cascade changes to the HDFS codebase, as an existing mechanism is used. This means no changes to the Datanode or code are needed, although it might be advantageous to extend the INode properties to indicate if a file is encrypted. Encryption and decryption are the responsibility of the client (HDFS or MapReduce), as well as key management and protection. On a potentially negative side, this means if encryption of all files is desired, it cannot be enforced at the HDFS level as in my design. Protection of keying material is not covered in the design guide; instead, they created the concept of a CryptoContextProvider. From

the design guide: “For example, a CryptoContextProvider implementation may utilize the Java KeyStore. In this case, additional parameters for specifying the type and location of the keystore must be provided in the job configuration when submitting the job. Please note that such parameters may be sensitive and it may not be appropriate to include them into a job configuration. Job configuration files are usually distributed to the cluster unprotected.” [50] An approach to implementing protected keys would be to create a CryptoContextProvider that stores keys within the cluster and protects them with the TPM via a binding key. An interesting approach might be one similar to the file-oriented design, where the NameNode gives the client a one-time AES key to use for the file that it binds to the TPM. A MapReduce job running on the NameNode would ask for this key from the NameNode and the Hadoop permissions would be enforced to determine if this key should be released.

7.3 Other Crypto Schemes

The presented design relies on a crypto-system that utilizes both asymmetric Public Key cryptography and symmetric key cryptography. The asymmetric encryption is primarily used as a RSA TPM key that is used as a wrapping key for faster AES keys that are used to encrypt and decrypt HDFS files. The AES key size can be configured for 128 bit or 256 bit depending on the nature of the application. Future work should include replacing the RSA key with an Elliptic Curve based key with the finalization and release of TPM 2.0, which will support it. Elliptic Curve keys have the advantage of having stronger encryption in a far fewer number of bytes than RSA in terms of key size. Other future work could consider exploring the use of fully homomorphic encryption as in the paper mentioned in the literature review [28]. A weakness in the current design is that

data must be decrypted to operate on, making it vulnerable while in memory during a MapReduce job. Homomorphic encryption has the promise to allow data to remain encrypted and processed upon by an encrypted query. Although homomorphic encryption has been around for some time, researchers at IBM have only recently released a low-level tool kit (almost assembly level) for working with homomorphic encryption schemes in polynomial time [52]. Given how low-level this tool kit is, it would be impractical to implement into Hadoop at this time, but as higher-level abstractions are developed, this will change. Also, although the algorithms work in polynomial time, the speed needs much improvement to make it practical [52]. Still, this idea is tantalizing and Hadoop would make a good use case for this style of encryption.

7.4 Future Work: Purposeful Attestation and Validation

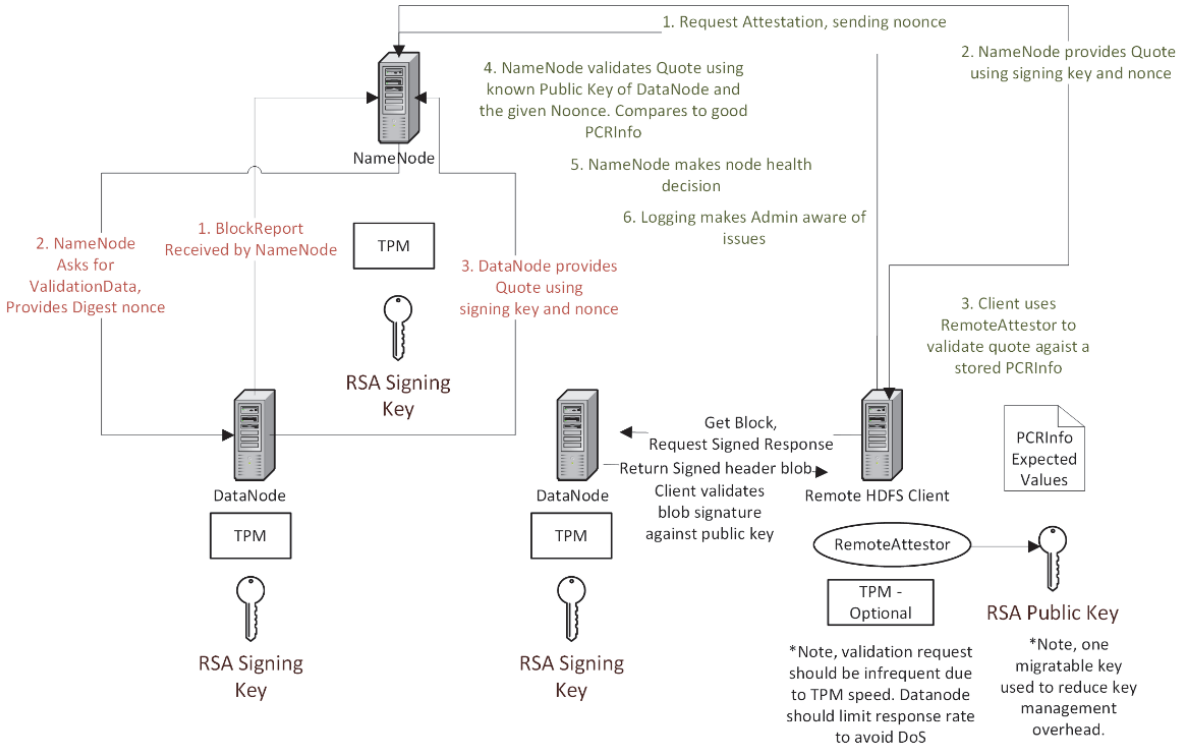

Figure 27. Purposeful Attestation Concept

The concept of using open platform trust services to conduct the base Hadoop OS platform attestation with possible integration into a Hadoop management health portal view was discussed. In addition, by using components of the Java trusted software stack provided by the JSR321 javax.trustedcomputing API and the jTSS implementation, we can create a purposeful attestation environment that could augment or replace this. In this case, we can use the application PCR (PCR 23), and optionally combine with any platform state PCRs such as the IMA PCR and TrustedGRUB PCRs to implement a custom attestation engine. Unlike the traditional TCG concept of attestation, which works via a privacy CA or Direct Anonymous Attestation, a purposeful attestation system needs no protection of privacy, and we build logic directly into the HDFS client, NameNode, and DataNode code to initiate an attestation. Since privacy is not a concern, there is no reason to use an AIK, as the public key part of a TPM signing key could be distributed to systems that will be partners in the attestation process. For instance, DataNodes could be required to report platform state directly to the NameNode on a predetermined window, perhaps in conjunction with the block report, or a NameNode or client could query DataNodes directly via the TSS being exposed via the network SOAP interface. The NameNode could then prevent a DataNode presenting an incorrect quote from participating in the cluster until the issue is resolved, directing client request to other nodes. This system would require the NameNode to maintain a protected database of known good values for the DataNodes in the cluster, as well as the public part of the RSA key that the DataNode would use to sign the PCR. The protocol would work by having a NameNode send a nonce to the DataNode. The DataNode, making use of the Javax.trustedcomputing.tpm.tools.Attestor package's quote method, creates a

ValidationData object that is based on a PCRindicies object, identity or signing key, and a Digest of the nonce received from the remote validator. Then the NameNode uses the Javax.trustedcomputing.RemoteAttestor package's validate-quote method and uses the public key part of the DataNode TPM's signing RSA key, a known PCRInfo object, and the original nonce to validate that the quote came from a known node and that it is in a known state based on the PCR values. Another simple option to validate that one is communicating with a known node is to add a signature exchange to the protocol. The DataNode would make use of the Javax.trustedcomputing.tpm.tools.Signer package and would sign a blob with the private TPM RSA key. A remote client would make use of the Javax.trustedcomputing.tpm.tools.remote.RemoteSigner package to validate the signed blob with the public part of the key.

CONCLUSION

This research presents a method of protecting big data and moving towards a more trusted Apache Hadoop HDFS infrastructure entailing identifying risk and threats, creating a base platform that is measured and verified by use of off-the-shelf TCG technologies, and implementing data protections via hardware-accelerated encryption with key protections tied to the hardware-based TPM. A mechanism by which the state of a platform can be verified from a Hadoop client prior to sending data to it via an attestation protocol is described. Testing results validated claims being made about encryption overhead reduction via the AES-NI instruction, with 16% overhead for encryption and 11% overhead for decryption being shown in testing of simulated 128MB block data. The speed of various TPM operations were also identified that would scale and apply to other solutions, and alternative designs were proposed to limit the overhead produced by the initial implementation, as well as potential ways to integrate with the recently proposed Hadoop crypto framework. Besides the direct applicability to Hadoop, the general steps taken to apply off-the-shelf TCG components to a highly application specific infrastructure can be taken to a number of other distributed data solutions. Integrating key protections and AES-NI accelerated encryption into other solutions can also be applied in similar fashions. For instance, MongoDB, an open source document database featuring a NoSQL design, is distributed in a similar fashion. Although written in C++, it also does not support data-at-rest encryption natively [53]. Implementing an encryption and key protection engine would be technically different at the code level, but could work similarly architecturally.

Data stored in Apache Hadoop can provide an attractive target for hackers, insiders, competitors, and nation-state adversaries. Implementing Trusted Computing concepts and utilizing technologies including the TPM in an IT infrastructure and in software design can provide an architecture with a solid security core, and provide a compelling boundary in a layered security design, moving towards a platform that can be trusted.

APPENDIX A: SOURCE CODE EXAMPLE – TPM UTILITIES USING JAVAX.TRUSTEDCOMPUTING

```
package tpmSoftwareIntegrationTest;

import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.interfaces.RSAPublicKey;
import java.util.UUID;

import javax.trustedcomputing.AbstractTestCase;
import javax.trustedcomputing.TrustedComputingException;
import javax.trustedcomputing.tpm.TPM;
import javax.trustedcomputing.tpm.TPMContext;
import javax.trustedcomputing.tpm.keys.BindingKey;
import javax.trustedcomputing.tpm.keys.KeyManager;
import javax.trustedcomputing.tpm.keys.StorageKey;
import javax.trustedcomputing.tpm.keys.StorageRootKey;
import javax.trustedcomputing.tpm.keys.TPMKey;
import javax.trustedcomputing.tpm.structures.Digest;
import javax.trustedcomputing.tpm.structures.PCRInfo;
import javax.trustedcomputing.tpm.structures.Secret;
import javax.xml.bind.Binder;
//
/**
 *@author Jason Cohen, Towson University
 *Note: Contains methods adapted from examples given in javax.trustedcomputing
TCK test package credit and rights as deserved to
 *IAIK, Graz University of Technology and thanks for your efforts towards a
usable TC API for Java.
 *
 */
public class TPMUtils {

	public TPMContext context_;

	public KeyManager keyManager_ = null;
	public BindingKey bindingKey = null;
	private StorageRootKey srk = null;
	private Secret srkSecret = null;


	public TPMUtils (){
		try {
			this.initializeTpmContext();
		} catch (Exception e) {

			e.printStackTrace();
		}

	}
	 protected void finalize()
	{
		try {
			context_.close();
		} catch (TrustedComputingException e) {
			// TODO Auto-generated catch block

		}
```

```
	}


	public static byte [] convertPassword (String encoding, String type,
String pw) throws UnsupportedEncodingException, NoSuchAlgorithmException {
//encoding is ACSII, UFT, etc, type is PLAIN or SHA1

	 if (type.equals("PLAIN")) {
			byte[] pwAsBytes = pw.getBytes(encoding);
			MessageDigest md = MessageDigest.getInstance("SHA-1");
			md.update(pwAsBytes);
			return md.digest();
	 } else if (type.equals("SHA1")) {
			byte[] asBytes = new byte[pw.length() / 2];
			for (int i = 0; i < asBytes.length; i++) {
				asBytes[i] = (byte) Short.parseShort(pw.substring(i *
2, i * 2 + 2), 16);
			 }
			return asBytes;
	 }
	 else return null;


	}


		public void initializeTpmContext() throws Exception {
			context_ =
TPMContext.getInstance("iaik.tc.jsr321.tpm.TPMContextImpl");
			context_.connect(null);
			if (context_.isConnected() == false) {
				System.err.println("TPM Context Failed");
			}
		}

	public void initializeKeyManager() throws Exception{
		if (this.context_==null || context_.isConnected() == false)
		{
			System.err.println("Error: Context not initialized in
getStorageKey, trying to initialize");
			this.initializeTpmContext();
		}
	keyManager_ = context_.getKeyManager();
	if (keyManager_== null) {
		System.err.println("KeyManager initialize failed");
	}

}
		public int loadStorageRootkey() throws Exception {
			if (this.context_==null || context_.isConnected() == false)
			{
				System.err.println("Error: Context not initialized in
loadStorageKey, trying to initialize");
				this.initializeTpmContext();
			}
			if (this.keyManager_==null)
			{

				this.initializeKeyManager();
			}
			if (this.srkSecret==null)
			{
				System.err.println("srkPassword Not Set");
```

```
                        return -1;
                }

                this.srk = keyManager_.loadStorageRootKey(this.srkSecret);
                return 0;
        }

        public StorageRootKey getStorageRootKey() throws Exception {
                if (this.context_==null || context_.isConnected() == false)
                {
                        System.err.println("Error: Context not initialized in
getStorageKey, trying to initialize");
                        this.initializeTpmContext();
                }
                if (this.keyManager_==null)
                {

                        this.initializeKeyManager();
                }
                if (this.srkSecret==null)
                {
                        System.err.println("srkPassword Not Set");
                        return null;
                }
                if (this.srkSecret==null)
                {
                        this.loadStorageRootkey();
                }
                return this.srk;

        }

        public void setSrkSecret (byte[] srkpw)
        {
                this.srkSecret =
context_.getSecret(context_.getDigest(srkpw));
        }

        public StorageKey createStorageKey( byte[] newKeyPw, byte []
migrationPW, boolean migratable, boolean isVolatile) throws Exception {

                if (this.srk==null)
                {
                        loadStorageRootkey();
                }

                Secret usageSecret = context_.getSecret(context_
                                .getDigest(newKeyPw));
                Secret migrationSecret = context_.getSecret(context_
                                .getDigest(migrationPW));

                StorageKey storageKey =
keyManager_.createStorageKey(this.srk, usageSecret,
                                migrationSecret, migratable, isVolatile, true,
null);

                if (storageKey == null)
                {


                        System.err.println("StorageKey was not created");
                        return null;
                }
```

```
                    System.out.println("Storage Key "+storageKey.getUUID()+ "
was created");
                    return storageKey;

             }

             public BindingKey createBindingKey(byte[] newKeyPw, byte []
migrationPW, boolean migratable, boolean isVolatile) throws Exception {
                    if (this.srk==null)
                    {
                           loadStorageRootkey();
                    }

                    Secret usageSecret = context_.getSecret(context_
                                  .getDigest(newKeyPw));
                    Secret migrationSecret = context_.getSecret(context_
                                  .getDigest(migrationPW));

                    BindingKey bindingKey =
keyManager_.createBindingKey(this.srk, usageSecret,
                                  migrationSecret, migratable, isVolatile, true,
2048, null);


                    if (bindingKey == null)
                    {

                           System.err.println("bindingKey was not created");
                           return null;
                    }
                    System.out.println("Storage Key "+bindingKey.getUUID()+ "
was created");
                    return bindingKey;

             }
             public  byte[] getRandom(int numBytes) throws
TrustedComputingException //32 bytes is size for a 256 bit key
             {
                    byte[] random = new byte[numBytes];
                    byte [] temp = context_.getTPMInstance().getRandom(numBytes
+20); //doing this because function does not guarantee exact number of bytes
requested will be returned
                    for (int i = 0; i<numBytes; i++)
                    {

                           random[i] = temp[i];
                    }
                    return random;
             }

             public byte[] bindData(BindingKey bindingKey, byte[]input) throws
Exception {
                    if (this.srk==null)
                    {
                           loadStorageRootkey();
                    }

                    javax.trustedcomputing.tpm.tools.Binder binder =
context_.getBinder();

                    byte[] boundData = binder.bind(input,
bindingKey.getPublicKey());
```

```
                    if (boundData.length==0)
                    {
                           System.err.println("error binding data with,
exiting");
                           System.exit(-1);
                    }
                    return boundData;


             }
             public byte[] unbindData(BindingKey bindingKey, byte[]input)
throws Exception {
                    if (this.srk==null)
                    {
                           loadStorageRootkey();
                    }

                    javax.trustedcomputing.tpm.tools.Binder binder =
context_.getBinder();

                    byte[] unboundData = binder.unbind(input, bindingKey);

                    if (unboundData.length==0)
                    {
                           System.err.println("error unbinding data with,
exiting");
                           System.exit(-1);
                    }
                    return unboundData;


             }
             public void storeKey(TPMKey key) throws Exception {
                    if (this.srk==null)
                    {
                           loadStorageRootkey();
                    }

                    keyManager_.storeTPMSystemKey(this.srk, key,
key.getUUID());
             }
             public void listStoredKeys() throws Exception
             {
                    if (this.keyManager_==null)
                    {

                           this.initializeKeyManager();
                    }
                    UUID [] keys = null;
                    keys = keyManager_.getStoredTPMSystemKeys();
                    for (int i=0;i<keys.length;i++)
                    {
                           System.out.println(keys[i].toString());
                    }

             }
             public BindingKey loadBindingKey(UUID u) throws Exception
             {
                    if (this.srk==null)
                    {
                           loadStorageRootkey();
                    }
```

```
                    BindingKey b = (BindingKey)
keyManager_.loadTPMSystemKey(this.srk, u, this.srkSecret);
                    return b;
             }
             public void setBindingKey (BindingKey b)
             {
                    this.bindingKey = b;
             }

             public void printPCRs() throws TrustedComputingException {
                    TPM tpm = context_.getTPMInstance();

                    int[] indices = new int[24];
            for (int i = 0; i < 24; i++) {
                    indices[i] = i;
            }

            PCRInfo p = tpm.readPCR(indices);

                  for (int i = 0; i < 24; i++) {
                     Digest d = null;
                         d = p.getPCRValue(i);
                 byte[] PCR = d.getBytes();
                 System.out.println("PCR "+i+":"+Utils.bytesToHex(PCR));
        }



             }
}
```

APPENDIX B: SOURCE CODE EXAMPLE – AES UTILITIES AND MICRO BENCHMARKS

```
package tpmSoftwareIntegrationTest;

import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.trustedcomputing.TrustedComputingException;
/*
 * @author Jason Cohen - Towson University
 * 2013
 */
public class AESUtils {

	public static SecretKeySpec generateAESKey(int keySize, boolean useTPM)
	// Note, SecretKeySpec is used for raw keys without using provider based
	// factory, since we don't need to support anything other then raw keys,
	// going with this method
	{
		byte[] random = null;
		if (useTPM == true) {
			try {
				random = new TPMUtils().getRandom(keySize/8);
			} catch (TrustedComputingException e) {
				// TODO Auto-generated catch block
				System.err
					.println("Unable to generate AES key
for  file, exiting");
				e.printStackTrace();
				System.exit(0);
			}
		} else {
			KeyGenerator keyGen;
			try {
				keyGen = KeyGenerator.getInstance("AES");
				keyGen.init(keySize);
				SecretKeySpec skey = (SecretKeySpec)
keyGen.generateKey();
				return skey;
			} catch (NoSuchAlgorithmException e) {
				System.err.println("AES Not Supported on this system,
exiting");
				System.exit(-1);
				e.printStackTrace();
			}
```

```
		}
		SecretKeySpec skey = new SecretKeySpec(random, "AES");
		return skey;
	}


	public static void encryptFileStream(FileInputStream fis,
			FileOutputStream fos, SecretKeySpec key) throws IOException
{
		Cipher cipher = null;
		try {
			cipher = Cipher.getInstance("AES");

		} catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.err.println("AES not Supported on system, exiting");
			System.exit(-1);
		}
		try {

			cipher.init(Cipher.ENCRYPT_MODE, key);
		} catch (InvalidKeyException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			e.printStackTrace();
			System.err.println("Some issue with generated key,
exiting");
			System.exit(-1);
		}
		CipherOutputStream cos = new CipherOutputStream(fos, cipher);

		byte[] block = new byte[131072]; //131072 is 128K 16384 is 16KB
65536 is 64K
		int i;
		while ((i = fis.read(block)) != -1) {
			cos.write(block, 0, i);
		}
		cos.flush();
		cos.close();
		fos.flush();
		fos.close();
	}


	public static void encryptFileStream(byte[] fis,
			FileOutputStream fos, SecretKeySpec key) throws
IOException, IllegalBlockSizeException, BadPaddingException {
		Cipher cipher = null;
		try {
			cipher = Cipher.getInstance("AES");

		} catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.err.println("AES not Supported on system, exiting");
			System.exit(-1);
		}
		try {

			cipher.init(Cipher.ENCRYPT_MODE, key);
		} catch (InvalidKeyException e) {
```

```
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                    e.printStackTrace();
                    System.err.println("Some issue with generated key,
exiting");
                    System.exit(-1);
             }
             long start = System.currentTimeMillis();
             byte [] encrypted = cipher.doFinal(fis);
             long stop = System.currentTimeMillis();
             System.out.println("Actual Encryption time was "+(stop - start)+ "
ms");

             BufferedOutputStream bos = new BufferedOutputStream(fos);
             bos.write(encrypted);
             bos.flush();
             bos.close();
             fos.close();
      }

      public static void encryptFileStream(byte[] fis,
                    FileOutputStream fos,  Cipher cipher) throws IOException,
IllegalBlockSizeException, BadPaddingException {

             long start = System.currentTimeMillis();
             byte [] encrypted = cipher.doFinal(fis);
             long stop = System.currentTimeMillis();
             System.out.println("Actual Encryption time was "+(stop - start)+ "
ms");

             BufferedOutputStream bos = new BufferedOutputStream(fos);
             bos.write(encrypted);
             bos.flush();
             bos.close();
             fos.close();
      }

      public static void decryptFileStream(FileInputStream fis,
      FileOutputStream fos, SecretKeySpec key) throws IOException {
             Cipher cipher = null;
             try {
                    cipher = Cipher.getInstance("AES");

             } catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                    System.err.println("AES not Supported on system, exiting");
                    System.exit(-1);
             }
             try {

            cipher.init(Cipher.DECRYPT_MODE, key);
             } catch (InvalidKeyException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                    e.printStackTrace();
                    System.err.println("Some issue with generated key,
exiting");
                    System.exit(-1);
             }

      CipherInputStream cis = new CipherInputStream(fis, cipher);
```

```
	byte[] block = new byte[65536]; //131072 is 128K 16384 is 16KB 65536 is
64K
	int i;
	while ((i = cis.read(block)) != -1) {
		fos.write(block, 0, i);
	}

	cis.close();
	fos.flush();
	fos.close();

	}


	public static void decryptFileStream(FileInputStream fis,
	FileOutputStream fos, Cipher cipher) throws IOException {


	CipherInputStream cis = new CipherInputStream(fis, cipher);

	byte[] block = new byte[65536]; //131072 is 128K 16384 is 16KB 65536 is
64K
	int i;
	while ((i = cis.read(block)) != -1) {
		fos.write(block, 0, i);
	}

	//cis.close();
	fos.flush();
	fos.close();

	}

	public static void decryptFileStream(byte[] fis,
	FileOutputStream fos, SecretKeySpec key) throws IOException,
IllegalBlockSizeException, BadPaddingException {
		Cipher cipher = null;
		try {
			cipher = Cipher.getInstance("AES");

		} catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.err.println("AES not Supported on system, exiting");
			System.exit(-1);
		}
		try {

		cipher.init(Cipher.DECRYPT_MODE, key);
		} catch (InvalidKeyException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			e.printStackTrace();
			System.err.println("Some issue with generated key,
exiting");
			System.exit(-1);
		}
		long start = System.currentTimeMillis();
		byte[] decrypted = cipher.doFinal(fis);
		long stop = System.currentTimeMillis();
		System.out.println("Actual Decryption time was "+(stop - start)+ "
ms");
```

```
	BufferedOutputStream bos = new BufferedOutputStream(fos);
	bos.write(decrypted);
	bos.flush();
	bos.close();
	fos.close();

	}


	public static CipherOutputStream getCipherOutputStream(FileOutputStream
fos, SecretKeySpec key, String Algorithm) throws IOException {
		Cipher cipher = null;

		try {
			cipher = Cipher.getInstance(Algorithm); //intention for
this implementation is AES only

		} catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.err.println("Encryption Algorithm not Supported on
system, exiting");
			System.exit(-1);
		}
		try {

			cipher.init(Cipher.ENCRYPT_MODE, key);
		} catch (InvalidKeyException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			e.printStackTrace();
			System.err.println("Some issue with generated key,
exiting");
			System.exit(-1);
		}
		CipherOutputStream cos = new CipherOutputStream(fos, cipher);
		return cos;


	}

	public static Cipher getCipher(SecretKeySpec key, String  mode, String
Algorithm, byte[] IV) throws InvalidKeyException //doing this seperate so we
have a easy way to get the random IV
	, NoSuchAlgorithmException, NoSuchPaddingException,
InvalidAlgorithmParameterException
		{
			Cipher cipher = null;

			cipher = Cipher.getInstance(Algorithm); //intention for
this implementation is AES only

				if (mode.equals("ENCRYPT_MODE")==true)
				{
				cipher.init(Cipher.ENCRYPT_MODE, key);
				}
				else {
					if (IV !=null){
					cipher.init(Cipher.DECRYPT_MODE, key, new
IvParameterSpec(IV));
					}
					else cipher.init(Cipher.DECRYPT_MODE, key);
				}
```

```
                        return cipher;

            }

      public static CipherOutputStream getCipherOutputStream(FileOutputStream
fos, Cipher cipher ) throws IOException {

            CipherOutputStream cos = new CipherOutputStream(fos, cipher);
            return cos;


      }



      public static CipherInputStream getCipherInputStream(InputStream in,
Cipher cipher) throws Exception {


            CipherInputStream cis = new CipherInputStream(in, cipher);
            return cis;


      }
}




package tpmSoftwareIntegrationTest;

import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Arrays;
import java.util.UUID;

import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.spec.SecretKeySpec;
import javax.trustedcomputing.*;
import javax.trustedcomputing.tpm.*;
import javax.trustedcomputing.tpm.keys.BindingKey;
import javax.trustedcomputing.tpm.keys.KeyManager;
import javax.trustedcomputing.tpm.keys.StorageRootKey;
import javax.trustedcomputing.tpm.structures.Digest;
import javax.trustedcomputing.tpm.structures.PCRInfo;
import javax.trustedcomputing.tpm.structures.Secret;



/**
 * @author Jason Cohen Towson University (c) 2013
 *
 *         code to test Trusted Computing API functions and encryption
 *         performance used in Hadoop project
 *
 */

public class APITest {
```

```
	public static void APITest1() {
		try {

			// test decrypting a block file
			String blockPath = "/dfs/dn/current/BP-357818861-
192.168.1.2-1356272950327/current/finalized/blk_-3212612316052151855";
			String outputFile = "/home/hdfs/decryptionTest.txt";

			byte[] keyIn1 = Utils.readFile(new File(blockPath +
"_20965.key"));
			byte[] keyPart1 = Arrays.copyOfRange(keyIn1, 0, 16);// the
key file

				// has both the

				// AES key and

				// IV, parse out

				// these parts
			byte[] IVPart1 = Arrays.copyOfRange(keyIn1, 16,
keyIn1.length);
			System.out.println("Converting key to SecretKeySpec length
is "
					+ (keyPart1.length * 8) + "bits" + " IV was "
					+ IVPart1.length + " bytes");

			SecretKeySpec key1 = new SecretKeySpec(keyPart1, "AES");
			System.out.println("SecretKeySpec created as a "
					+ key1.getAlgorithm());
			try { // NOTE, using ECB for now, will use CBC later and
save IV to
					// the key file and parse
				Cipher cipher = AESUtils.getCipher(key1,
"DECRYPT_MODE",
						"AES/CBC/PKCS5PADDING", IVPart1);
				FileInputStream blockIn = new FileInputStream(new
File(
						blockPath));
				CipherInputStream blockInDecrypt = AESUtils
						.getCipherInputStream(blockIn, cipher);
// get a

				// CipherInputStream

				// to wrap the

				// Block stream
				DataInputStream blockInDS = new
DataInputStream(blockInDecrypt);
				FileOutputStream blockDecryptOut = new
FileOutputStream(
						new File(outputFile));
				int c;
				while ((c = blockInDS.read()) != -1) {
					blockDecryptOut.write(c);
				}
				blockDecryptOut.close();
			} catch (Exception e) {
				e.printStackTrace();
			}
```

```
                    System.exit(0);

                    // done decryption test
                    // test random AES key generation using TPM
                    long start = System.currentTimeMillis();
                    SecretKeySpec key = AESUtils.generateAESKey(128, true);
                    // System.out.println(key.getEncoded().length + " "+
                    // key.getFormat()+ " "+ new String(key.getEncoded()));

                    long stop = System.currentTimeMillis();
                    System.out.println("TPM AES generation Operation took "
                                + (stop - start) + " ms");

                    // test random AES key generation using Java
                    start = System.currentTimeMillis();
                    // SecretKeySpec key2 = new AESUtils().generateAESKey(128,
false);
                    // System.out.println(key2.getEncoded().length + " "+
                    // key2.getFormat()+ " "+ new String(key2.getEncoded()));
                    stop = System.currentTimeMillis();
                    System.out.println("Java keygen Operation took " + (stop -
start)
                                + " ms");

                    String path = "/home/WestburyLab.wikicorp.201004.txt";
                    String pathEnc =
"/home/WestburyLab.wikicorp.201004.txt.enc";
                    // Test simply writing the inputstream to a new file (for
                    // comparison)
                    /*
                     * start = System.currentTimeMillis(); FileInputStream fis2
= new
                     * FileInputStream(path); Utils.writeFile(fis2,
                     * path+System.currentTimeMillis()+".txt"); stop =
                     * System.currentTimeMillis();
                     * System.out.println("Copy Operation took "+ (stop-start)
+ " ms");
                     */
                    // Test encrypting File InputStream
                    File file = new File(path);
                    // if file doesn't exists, then create it
                    if (!file.exists()) {
                          file.createNewFile();
                    }
                    File fileEnc = new File(pathEnc);
                    // if file doesn't exists, then create it
                    if (!fileEnc.exists()) {
                          fileEnc.createNewFile();
                    } else {
                          fileEnc.delete(); // not sure why, but if this file
already
                                                              // exists, the
encryption operation is very
                                                              // slow
                    }
                    FileInputStream fis = new FileInputStream(file);
                    FileOutputStream fos = new FileOutputStream(pathEnc);
                    start = System.currentTimeMillis();
                    new AESUtils().encryptFileStream(fis, fos, key);
                    stop = System.currentTimeMillis();
                    System.out.println("Encryption Operation took " + (stop -
start)
                                + " ms");
```

```
                    fis.close();

                    // Test encrypting File Loaded from Memory
                    /*
                     * File fileEnc2 = new File(pathEnc); // if file doesn't
exists,
                     * then create it if (!fileEnc2.exists()) {
                     * fileEnc2.createNewFile(); } else { fileEnc2.delete(); //
not sure
                     * why, but if this file already // exists, the encryption
operation
                     * is very // slow }
                     *
                     * byte[] plaintxt = Utils.readFile(new File(path));
                     * FileOutputStream fos4 = new FileOutputStream(pathEnc);
start =
                     * System.currentTimeMillis(); new
                     * AESUtils().encryptFileStream(plaintxt, fos4, key); stop
=
                     * System.currentTimeMillis();
                     * System.out.println("Encryption from Memory Operation
took " +
                     * (stop - start) + " ms"); fis.close();
                     */
                    // Test Decrypting a file

                    FileInputStream fis3 = new FileInputStream(pathEnc);
                    FileOutputStream fos2 = new FileOutputStream(path +
"decrypted"
                                    + System.currentTimeMillis());
                    start = System.currentTimeMillis();
                    new AESUtils().decryptFileStream(fis3, fos2, key);
                    stop = System.currentTimeMillis();
                    System.out.println("Decryption Operation took " + (stop -
start)
                                    + " ms");
                    fis.close();

                    // Test Decrypting a file loaded to memory first
                    /*
                     * FileOutputStream fos5 = new FileOutputStream(path +
"decrypted" +
                     * System.currentTimeMillis()); byte[] infile =
Utils.readFile(new
                     * File(pathEnc)); start = System.currentTimeMillis(); new
                     * AESUtils().decryptFileStream(infile, fos5, key); stop =
                     * System.currentTimeMillis();
                     * System.out.println("Decryption from Memory Operation
took " +
                     * (stop - start) + " ms");
                     */
                    // Bind the AES KEY to a TPM Key
                    start = System.currentTimeMillis();
                    /* Create a TPMUtil Object and set it up for use */
                    TPMUtils u = new TPMUtils();
                    u.initializeKeyManager();
                    u.setSrkSecret(u.convertPassword("ASCII", "PLAIN",
"Password1"));
                    u.loadStorageRootkey();

                    /* Test Creating a Binding key */
                    start = System.currentTimeMillis();
                    BindingKey b = u.createBindingKey(
```

```
                                u.convertPassword("ASCII", "PLAIN",
"Password1"),
                                u.convertPassword("ASCII", "PLAIN",
"Password1"), true,
                                true);
                    stop = System.currentTimeMillis();
                    System.out.println("Creating a Binding Key Operation took "
                                + (stop - start) + " ms");
                    /* Test Storing Binding Key */
                    start = System.currentTimeMillis();
                    u.storeKey(b);
                    stop = System.currentTimeMillis();
                    System.out.println("Storing Binding Key Operation took "
                                + (stop - start) + " ms");
                    u.listStoredKeys();
                    /* Test Deleting a Binding Key */
                    u.keyManager_.deleteTPMSystemKey(b.getUUID());
                    u.listStoredKeys();

                    /*
                     * Load a Key from a UUID and set it be the Binding key we
use for
                     * encrypting our AES keys
                     */
                    start = System.currentTimeMillis();
                    UUID keyid = UUID
                                .fromString("294146af-1dd5-4b6f-b198-
0a4b87153c6e");
                    u.setBindingKey(u.loadBindingKey(keyid));
                    stop = System.currentTimeMillis();
                    System.out.println("Loading a Binding Key Operation took "
                                + (stop - start) + " ms");

                    /* Test Encrypting our AES Key to a TPM Binding Key */

                    start = System.currentTimeMillis();
                    byte[] encryptedKey = u.bindData(u.bindingKey,
key.getEncoded());
                    Utils.writeFile(encryptedKey, new
FileOutputStream("/root/key.enc"));
                    stop = System.currentTimeMillis();
                    System.out.println("Binding AES Key Operation took "
                                + (stop - start) + " ms");

                    /* Test Unbinding our AES key (Decrypting) */

                    start = System.currentTimeMillis();
                    System.exit(0);
                    // got the AES key again, now decryt the data
                    FileOutputStream fos6 = new FileOutputStream(path
                                + "decrypted2-.txt");
                    byte[] infile = Utils.readFile(new File(pathEnc));
                    long start2 = System.currentTimeMillis();
                    byte[] encryptedKeyFile = Utils.readFile(new
File("/root/key.enc"));
                    byte[] decryptedKeyBytes = u.unbindData(u.bindingKey,
                                encryptedKeyFile);
                    SecretKeySpec decryptedKey = new
SecretKeySpec(decryptedKeyBytes,
                                "AES");
                    new AESUtils().decryptFileStream(infile, fos6,
decryptedKey);
                    stop = System.currentTimeMillis();
```

```
                    System.out
                                .println("UnBinding AES Key Operation and
decryption of file took "
                                        + (stop - start) + " ms");
                    System.out
                                .println("UnBinding AES Key Operation and
decryption of file not counting file load time took "
                                        + (stop - start2) + " ms");

                    /*
                     * Another AES test with Padding and IV
                     */
                    start = System.currentTimeMillis();

                    SecretKeySpec newKey = AESUtils.generateAESKey(128, true);
                    Cipher cipher = AESUtils.getCipher(newKey, "ENCRYPT_MODE",
                                "AES/CBC/PKCS5Padding", null);
                    byte[] IV = cipher.getIV();
                    byte[] in = Utils.readFile(new File("/home/testdata.txt"));
                    FileOutputStream encOut = new
FileOutputStream("/home/encTest.enc");
                    start = System.currentTimeMillis();
                    AESUtils.encryptFileStream(in, encOut, cipher);
                    FileOutputStream keyOut = new
FileOutputStream("/home/encTest.key");
                    keyOut.write(newKey.getEncoded());
                    keyOut.write(IV);
                    keyOut.close();
                    stop = System.currentTimeMillis();
                    System.out
                                .println("Hadoop style block file encryption
of file took "
                                        + (stop - start) + " ms");

                    // Ok, done writting a file, test decryption
                    start = System.currentTimeMillis();
                    byte[] keyIn = Utils.readFile(new
File("/home/encTest.key"));
                    byte[] keyPart = Arrays.copyOfRange(keyIn, 0, 16);// the
key file

                           // has both the

                           // AES key and

                           // IV, parse out

                           // these parts
                    byte[] IVPart = Arrays.copyOfRange(keyIn, 16,
keyIn.length);
                    SecretKeySpec dcKey1 = new SecretKeySpec(keyPart, "AES");
                    FileOutputStream fos8 = new
FileOutputStream("/home/decrypted.txt");
                    Cipher cipher1 = AESUtils.getCipher(dcKey1, "DECRYPT_MODE",
                                "AES/CBC/PKCS5Padding", IVPart);
                    AESUtils.decryptFileStream(
                                new FileInputStream("/home/encTest.enc"),
fos8, cipher1);
                    stop = System.currentTimeMillis();
                    System.out
                                .println("Hadoop style block file decryption
of file took "
                                        + (stop - start) + " ms");
```

```
                        // Hadoop Data Test

                        start = System.currentTimeMillis();

                        keyIn = Utils
                                    .readFile(new File(
                                                "/dfs/dn/current/BP-357818861-
192.168.1.2-1356272950327/current/finalized/blk_-
7211259507382032132_20633.key"));
                        keyPart = Arrays.copyOfRange(keyIn, 0, 16);// the key file

                // has both the

                // AES key and

                // IV, parse out

                // these parts
                        IVPart = Arrays.copyOfRange(keyIn, 16, keyIn.length);

                        SecretKeySpec dcKey = new SecretKeySpec(keyPart, "AES");
                        FileOutputStream fos7 = new FileOutputStream(
                                    "/home/decryptedHadoopData2.txt");
                        Cipher cipher11 = AESUtils.getCipher(dcKey, "DECRYPT_MODE",
                                    "AES/CBC/PKCS5Padding", IVPart);
                        AESUtils.decryptFileStream(
                                    new FileInputStream(
                                                "/dfs/dn/current/BP-357818861-
192.168.1.2-1356272950327/current/finalized/blk_-7211259507382032132"),
                                    fos7, cipher11);
                        stop = System.currentTimeMillis();
                        System.out.println("Hadoop block file decryption of file
took "
                                    + (stop - start) + " ms");

                        System.gc();

                        // u.printPCRs();

                } catch (javax.trustedcomputing.TrustedComputingException te) {
                        te.printStackTrace();

                } catch (Exception e) {
                        e.printStackTrace();

                }
        }

}
```

REFERENCES

[1] P. Bean and R. Barth, "Who is really using Big Data," Harvard Business Review, 9 2012. [Online]. Available: http://blogs.hbr.org/cs/2012/09/whos_really_using_big_data.html. [Accessed 7 2013].

[2] Apache, "HDFS Architecture Guide," 8 May 2012. [Online]. Available: http://hadoop.apache.org/common/docs/stable/hdfs_design.html. [Accessed 15 Jun 2012].

[3] K. Z. S. R. Owen O'Malley, "Hadoop Security Design," Yahoo!, 2009. [Online]. Available: carfield.com.hk:8080/document/.../hadoop-security-design.pdf. [Accessed 1 11 2012].

[4] P. Russom, "Hadoop Functionality that Needs Improvement," 04 2013. [Online]. Available: http://tdwi.org/Blogs/Philip-Russom/2013/04/Hadoop-Functionality-that-Needs-Improvement.aspx.

[5] J. Cohen and A. Subrata, "Towards a More Secure Apache Hadoop HDFS Infrastructure," in *7th International Conference, NSS 2013*, Madrid, Spain, 2013.

[6] J. C. Cohen and A. Subrata, "Incorporating hardware trust mechanisms in Apache Hadoop," 2012.

[7] Intel Corporation, "Fast, Low-Overhead Encryption for Apache Hadoop," [Online]. Available: https://hadoop.intel.com/pdfs/IntelEncryptionforHadoopSolutionBrief.pdf. [Accessed 7 2013].

[8] Mandiant corporporation, "Mandiant M-Trends," 2010. [Online]. Available: http://www.princeton.edu/~yctwo/files/readings/M-Trends.pdf . [Accessed 1 Nov 2012].

[9] T. White, Hadoop The Definitive Guide, Sebastopol: O'Reilly, 2010.

[10] Trusted Computing Group, "TCG Specification Architecture Overview V. 1.4," 2 Aug 2007. [Online]. Available: http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf. [Accessed 15 Jun 2012].

[11] D. Challener, K. Yoder, R. Catherman, D. Safford and L. V. Doorn, A Practical Guide to Trusted Computing, Crawfordsville: IBM Press, 2007.

[12] M. ,. D. T. T. A. Ryan, "Trusted Computing: TCG proposals," 4 Nov 2006. [Online]. Available: http://www.cs.bham.ac.uk/~mdr/teaching/modules/security/lectures/TrustedComputingTCG.html. [Accessed 15 Jun 2012].

[13] H. L. J. ,. a. Schmitz, "TPM-SIM: A Framework for Performance Evaluation of Trusted Platform Modules," 48th Design Automation Conference, Jun 2011. [Online]. Available: http://www.cs.binghamton.edu/~dima/dac11.pdf. [Accessed 15 Jun 2012].

[14] Trusted Computing Group, "TPM Main Part 1 Design Principles Specification Version 1.2 R 116," 1 Mar 2011. [Online]. Available: http://www.trustedcomputinggroup.org/files/static_page_files/72C26AB5-1A4B-B294-D002BC0B8C062FF6/TPM%20Main-Part%201%20Design%20Principles_v1.2_rev116_01032011.pdf. [Accessed 15 Jun 2012].

[15] J. Wiens, "A Tipping Point for the Trusted Platform," 6. [Online]. Available: http://www.informationweek.com/security/encryption/a-tipping-point-for-the-trusted-platform/208800939. [Accessed 8 11 2012].

[16] D. Palmer, "Understanding Trusted Computing From The Ground Up," [Online]. Available: http://electronicdesign.com/content/content/74661/74661_fig2.jpg. [Accessed 3 2014].

[17] "TrouSerS FAQ," [Online]. Available: http://trousers.sourceforge.net/faq.html. [Accessed 15 Jun 2012].

[18] "TrustedGRUB Documentation," Sirrix AG security technologies, [Online]. Available: http://projects.sirrix.com/trac/trustedgrub/wiki/Documentation. [Accessed 15 Jun 2012].

[19] "Linx IMA Wiki," 18 May 2012. [Online]. Available: http://sourceforge.net/apps/mediawiki/linux-ima/index.php?title=Main_Page. [Accessed 15 Jun 2012].

[20] "The return of EVM," 30 Jun 2010. [Online]. Available: http://lwn.net/Articles/394170/. [Accessed 15 Jun 2012].

[21] M. Zohar, "EVM," 24 Jun 2010. [Online]. Available: http://lwn.net/Articles/393673/. [Accessed 15 Jun 2012].

[22] S. Munetoh, "Open Platform Trust Services," IBM, 16 May 2011. [Online]. Available: http://iij.dl.sourceforge.jp/openpts/51879/userguide-0.2.4.pdf. [Accessed 15 Jun 2012].

[23] "IAIK jTSS - TCG Software Stack for the Java (tm) Platform," 16 Sep 2011. [Online]. Available: http://trustedjava.sourceforge.net/index.php?item=jtss/readme. [Accessed Jun 2012].

[24] R. Toegl, "JSR321 Wiki," Graz University of Technology , July 2012. [Online]. Available: https://java.net/projects/jsr321/pages/GettingStartedGuide. [Accessed 7 2013].

[25] Z. Quan, D. Xiao, C. Tang and C. Rong, "TSHC: Trusted Scheme for Hadoop Cluster," *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on Digital Object Identifier:,* vol. 10.1109/EIDWT.2013.66 , p. 344–349, 2013.

[26] A. Ruan and A. Martin, "TMR: Towards a Trusted MapReduce Infrastructure," *Services (SERVICES), 2012 IEEE Eighth World Congress on Trusted Services,* vol. 10.1109/SERVICES.2012.28 , p. 141–148, 2012.

[27] S. Park and Y. Lee, "Secure Hadoop with Encrypted HDFS," in *Grid and Pervasive Computing*, Seoul, 2013.

[28] S. Jin, S. Yang, X. Zhu and H. Yin, "Design of a Trusted File System Based on Hadoop," *Trustworthy Computing and Services Communications in Computer and Information Science ,* vol. 320, pp. 673-680, 2013.

[29] Y. Xin, Q. Shen, . Yang and S. Qing, "A Way of Key Management in Cloud Storage Based on Trusted Computing," *Network and Parallel ComputingLecture Notes in Computer Science,* vol. 6985, pp. 135-145, 2011.

[30] Apache, "All you wanted to know about Hadoop, but were too afraid to ask: genealogy of elephants.," 2 2012. [Online]. Available: https://blogs.apache.org/bigtop/entry/all_you_wanted_to_know. [Accessed 1 11 2012].

[31] Cloudera, "Clouddera CDH4 Security Guide," [Online]. Available: https://ccp.cloudera.com/download/attachments/21438266/CDH4_Security_Guide_4.1.pdf?version=3&modificationDate=1349900837000. [Accessed 23 11 2012].

[32] E. Olougouna, "Encryption in SMB 3.0," 10 2012. [Online]. Available: http://blogs.msdn.com/b/openspecification/archive/2012/10/05/encryption-in-smb-3-0-a-protocol-perspective.aspx . [Accessed 1 11 2012].

[33] D. Das, "Role of Delegation Tokens in Apache Hadoop Security," 08 2011. [Online]. Available: http://hortonworks.com/blog/the-role-of-delegation-tokens-in-apache-hadoop-security/. [Accessed 21 11 2012].

[34] R. Grimes, "Don't count on Kerberos to thwart pass-the-hash attack," [Online]. Available: http://www.infoworld.com/d/security-central/dont-count-kerberos-thwart-pass-the-hash-attacks-871?page=0,2. [Accessed 22 11 2012].

[35] A. Becherer, "Hadoop Security Design: Just add Kerberos? Really?," iSEC Partners, Inc, 2010.

[36] "HDFS-945 Issue Report," [Online]. Available: https://issues.apache.org/jira/browse/HDFS-945. [Accessed 20 12 2012].

[37] S. Venugopalan, "With Sentry, Cloudera Fills Hadoop's Enterprise Security Gap," Cloudera, 24 July 2013. [Online]. Available: http://blog.cloudera.com/blog/2013/07/with-sentry-cloudera-fills-hadoops-enterprise-security-gap/. [Accessed 1 11 2013].

[38] "Java Buffer Overflows," [Online]. Available: http://stackoverflow.com/questions/479701/does-java-have-buffer-overflows. [Accessed 12 11 2012].

[39] C. McCabe, "Harden edit log loader against malformed or malicious input," 23 03 2012. [Online]. Available: https://issues.apache.org/jira/browse/HDFS-3134. [Accessed 12 11 2012].

[40] US Gov, "National Vulnerability Database," 30 11 2012. [Online]. Available: http://web.nvd.nist.gov/view/vuln/search-results?query=Hadoop&search_type=all&cves=on . [Accessed 30 11 2012].

[41] Hewlett Packard, "2011 Full Year Cyber Security Risk Report," 9 2011. [Online]. Available: http://www.hpenterprisesecurity.com/collateral/report/CyberSecurityRisksReport.pdf. [Accessed 1 10 2012].

[42] J. Furrier, "Big Data Market is Big Business," 17 02 2012. [Online]. Available: http://www.forbes.com/sites/siliconangle/2012/02/17/big-data-is-big-market-big-business/. [Accessed 1 11 2012].

[43] O. O'Malley, "Motivations for Hadoop Security," 8 2011. [Online]. Available: http://hortonworks.com/blog/motivations-for-apache-hadoop-security/. [Accessed 04 12 2012].

[44] Microsoft, "Windows 8 Secured and Mesured Boot," [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/br259097.aspx. [Accessed 15 Jun 2012].

[45] C. Nie, "Dynamic Root of Trust in Trusted Computing," Oct 2007. [Online]. [Accessed 15 Jun 2012].

[46] R. Wojtczuk and J. Rutkowska, ""Press Cheat Sheet" for the Attacking Intel Trusted Execution Technology," Feb 2009. [Online]. Available: http://invisiblethingslab.com/press/itl-press-2009-02.pdf. [Accessed 15 Jun 2012].

[47] IBM, "Securing sensitive files with TPM keys," IBM, [Online]. Available: http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=%2Fliaai%2Ftpm%2Fliaaitpmstart.htm. [Accessed 15 Jun 2012].

[48] Intel , "Intel® AES-NI Performance Testing on Linux/Java Stack," 6 2012. [Online]. Available: http://software.intel.com/en-us/articles/intel-aes-ni-performance-testing-on-linuxjava-stack#jdk-performance-improvement-operations-per-minute-evaluation. [Accessed 2012 4 12].

[49] A. Ignaten, "Improved Advanced Encryption Standard (AES) Crypto performance on Java* with NSS using Intel® AES-NI Instructions," Intel, 04 2012. [Online]. Available: http://software.intel.com/en-us/articles/improved-advanced-encryption-standard-aes-crypto-performance-on-java-with-nss-using-intel. [Accessed 7 2013].

[50] L. Yi, C. Haifeng, L. Tianyou, A. Purtell, Z. Xiang and B. Antony, "Hadoop Crypto Design," 27 Feb 2013. [Online]. Available: https://issues.apache.org/jira/secure/attachment/12571116/Hadoop%20Crypto%20Design.pdf.

[51] m. noll, "benchmarking and stress testing an hadoop cluster," 9 4 2011. [Online]. Available: http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench/. [Accessed 1 11 2013].

[52] IBM Watson Lab, "Cryptography Research - Homomorphic Encryption," 20 5 2014. [Online]. Available: http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2661.

[53] MongoDB, "Introduction to MongoDB," 2013. [Online]. Available: http://www.mongodb.org/about/introduction/. [Accessed 1 5 2014].

[54] A. Thurai and Intel Corp, "Bid Data - Big Risk..?," 1 July 2013. [Online]. Available: http://communities.intel.com/community/datastack/blog/2013/07/01/big-data-big-risk .

[55] A. Shostack, "STRIDE Chart," 2007. [Online]. Available: http://blogs.msdn.com/b/sdl/archive/2007/09/11/stride-chart.aspx. [Accessed 12 11 2012].

[56] O. O'Malley, "Hadoop Security," Yahoo!, 2010. [Online]. Available: http://www.slideshare.net/hadoopusergroup/1-hadoop-securityindetailshadoopsummit2010. [Accessed 15 Jun 2012].

[57] N. Jain, "Hadoop - How it manages security," Nov 28 2011. [Online]. Available: http://clustermania.blogspot.com/2011/11/hadoop-how-it-manages-security.html . [Accessed 15 Jun 2012].

CURRICULUM VITAE

Name: Jason C. Cohen

[redacted]

Program of Study: Information Technology
Degree and Date To Be Conferred: Doctor of Science., 2014

Collegiate Institutions Attended:

- Towson University, 2007-2014, Doctor of Science, Information Technology, May 2014
- Towson University, 2003-2006, Master of Science, Applied Information Technology, August 2006
- Goucher College, 1999-2003, Bachelor of Arts, Major: Computer Science Minor: Cognitive Science, May 2003

Professional Publications:

- J. Cohen, S. Acharya "Towards a Trusted HDFS Storage Platform: Mitigating Threats to Hadoop Infrastructures Using Hardware-Accelerated Encryption with TPM-Rooted Key Protection" Elsevier Journal of Information Security and Applications Special Issue, July 2014 (accepted).

- J. Cohen, S. Acharya "Improving Trustworthiness of the Apache Hadoop Storage Platform: Design Considerations of Trusted Computing Based Threat Mitigations and Performance Metrics of an AES Based Encryption Scheme with TPM Rooted Key Protection", in the 10th IEEE International Conference on Autonomic and Trusted Computing (ATC-2013), Vietri sul Mare, Italy, December, 2013.

- J. Cohen, S. Acharya, "Towards a More Secure Apache Hadoop HDFS Infrastructure: Anatomy of a Targeted Advanced Persistent Threat against HDFS and Analysis of Trusted Computing Based Countermeasures", in the 7th International Conference on Network and System Security, (NSS 2013), Publisher: Springer Lecture Notes in Computer Science, June, 2013.

- J. Cohen, S. Acharya, "Defending Apache Hadoop with Trusted Computing", in LCA 2013 - Linux Conference Australia, Advances in Linux Security Mini conference, Canberra, Australia, January, 2013. http://mirror.linux.org.au/linux.conf.au/2013/mp4/Trusted_Computing_and_Hadoop_HDFS.mp4).

- J. Cohen, S. Acharya, "Incorporating Hardware Trust Mechanisms in Apache Hadoop", in the IEEE GlobeCom 2012 First International workshop on Management and Security technologies for Cloud Computing, ManSec-CC, Anaheim, California, USA, December, 2012.

Professional Positions Held:

- Technology Consultant, Hewlett Packard Company, Hanover, MD, 2007-Present
- Computer Scientist, Science Applications International Corporation, Columbia, MD, 2005-2007
- Systems Analyst, Lockheed Martin, Woodlawn, MD 2002-2005