DECENTRALIZING TRUST:
NEW SECURITY PARADIGMS FOR CLOUD COMPUTING

by

Safwan Mahmud Khan

APPROVED BY SUPERVISORY COMMITTEE:

Dedicated to my parents and wife,

for their endless encouragement, support and love.

Special dedication goes to my advisor, Dr. Kevin W. Hamlen.

DECENTRALIZING TRUST:
NEW SECURITY PARADIGMS FOR CLOUD COMPUTING

by

SAFWAN MAHMUD KHAN, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2013

ACKNOWLEDGMENTS

I would like to give my wholehearted appreciation and deepest gratitude to Dr. Kevin W. Hamlen for his assistance and relentless guidance in his supervision of my research efforts. It is probably fair to say, I would not have started my Ph.D. without his support, finished it without his direction, or enjoyed it as much as I did without his unparalleled knowledge and wonderful personality. Dr. Hamlen worked tirelessly, reviewing my research approach and providing many brilliant ideas on how to achieve successful qualitative research. Special thanks go to Dr. Murat Kantarcioglu for taking the initiative of applying language-based security in clouds, the last part of my dissertation.

This research was supported in part by the National Science Foundation under Grant No. NSF-0959096 and NSF-1065216, and the U.S. Air Force Office of Scientific Research Grant No. FA9550-10-1-0088. I want to acknowledge them here. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the NSF or AFOSR.

I would like to thank Dr. Latifur Khan, Dr. Murat Kantarcioglu and Dr. Kamil Sarac for their interest in my research and being part of my dissertation committee.

I spent more than four years here, and some of my friends really made my UTD experience. I wish to show appreciation to Altaf, Mehedy, Arindham, Sheikh, Satyen, Vaibhav, Mehmet, Erman, Huseyin, Meera, Frederico and Wenhao among others, for making my hectic research life easier and healthier. I will miss your friendly company and cooperation.

I want to especially acknowledge my wife, Farjana Sikder, for her continuous support, understanding, and love that provided the emotional environment that allowed me to go through the Ph.D. program and finish the dissertation. I am and will be always grateful to you. Special mention should go to my father-in-law who inspired me a lot to start my Ph.D.

Finally, I would like to thank Nesar Uddin Khan, my father, and Sufia Khanam, my mother, for the sacrifices they made to give me the opportunity to follow my passions, and for, in the end, making it possible for me to complete it successfully. You, and your love, are always the key inspiration for me behind everything. I am also thankful to my elder sister Tonni and younger brother Sheedy for being with me constantly.

Lastly, and most importantly, I am thankful to Almighty Allah for everything.

October 2013

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the “Guide for the Preparation of Master’s Theses and Doctoral Dissertations at The University of Texas at Dallas.” It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student’s contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

DECENTRALIZING TRUST:

NEW SECURITY PARADIGMS FOR CLOUD COMPUTING

Publication No. ____________________

Safwan Mahmud Khan, PhD
The University of Texas at Dallas, 2013

Supervising Professor: Dr. Kevin W. Hamlen

Data and computation integrity and security are major concerns for users of cloud computing facilities. Today's clouds typically place centralized, universal trust in all the cloud's nodes. This simplistic, full-trust model has the negative consequence of amplifying potential damage from node compromises, leaving such clouds vulnerable to myriad attacks.

To address this weakness, this dissertation proposes and evaluates new paradigms for decentralizing cloud trust relationships for stronger cloud security. Five new paradigms are examined: (1) *Hatman* decentralizes trust through computation replication in clouds to ensure computation integrity. Its prototype implementation embodies the first full-scale, data-centric, reputation-based trust management system for Hadoop clouds. (2) *AnonymousCloud* decentralizes trust by decoupling private billing information from a cloud job's code and data. The resulting cloud conceals computation and data ownership information from nodes that compute using the data, thereby impeding malicious nodes from learning who owns these resources without disrupting the cloud's power to process and bill jobs. (3) *Penny* is a fully decentralized peer-to-peer structure that shifts trust away from traditional, centralized, cloud master nodes to an equal distribution of trust over all nodes. It supports integrity and confidentiality labeling of data, and enforces a notion of ownership privacy that permits peers to publish data without revealing their ownership of the data. (4) *CloudCover* decentralizes trust on the user side. It proposes a new form of Security as a Service (SECaaS) that allows untrusted, mostly serial computations in untrusted computing environments to be independently and efficiently validated by trusted, commodity clouds. Finally, (5) *SilverLine* automatically in-lines secure information flow tracking code into untrusted job binaries, facilitating enforcement of custom security policies without any change to the underlying cloud kernel, operating system, hypervisor, or file system implementations. This makes SilverLine exceptionally easy to deploy and maintain on rapidly evolving cloud frameworks, since the cloud and security enforcement implementations are completely separate and orthogonal.

Experiments demonstrate that each paradigm is an effective strategy for realizing stronger security in cloud computing frameworks at modest overheads through reducing or shifting the trusted computing base.

TABLE OF CONTENTS

ACKNOWLEDGMENTS v
PREFACE vii
ABSTRACT viii
LIST OF FIGURES xiii
LIST OF TABLES xv
CHAPTER 1 INTRODUCTION 1
CHAPTER 2 HATMAN: INTRA-CLOUD TRUST MANAGEMENT FOR HADOOP 7
2.1 System Overview 9
2.1.1 Overview of Hadoop Architecture 9
2.1.2 Hatman Architecture 10
2.1.3 Activity Types 14
2.1.4 Attacker Model and Assumptions 15
2.2 Implementation 16
2.3 Results and Analysis 17
CHAPTER 3 ANONYMOUSCLOUD: A DATA OWNERSHIP PRIVACY PROVIDER FRAMEWORK IN CLOUD COMPUTING 21
3.1 System Architecture 22
3.1.1 Cloud Providers 23
3.1.2 Managers 24
3.1.3 Authentication Protocol 24
3.2 Results and Analysis 27
CHAPTER 4 PENNY: SECURE, DECENTRALIZED DATA MANAGEMENT . . 31
4.1 Penny Network 33
4.1.1 Definitions 34
4.1.2 Network Architecture 36

4.2 Implementation and Results of Experimental Evaluation 46

4.2.1 Bounding Neighborhood Size 47

4.2.2 Results for File Downloads 50

4.2.3 Results for RDF Datasets 55

4.3 Discussion 56

CHAPTER 5 COMPUTATION CERTIFICATION AS A SERVICE IN THE CLOUD 62

5.1 System Overview 65

5.1.1 Architecture and Threat Model 65

5.1.2 Computation Integrity Proof Generation and Validation 67

5.1.3 CloudCover Protocol 69

5.1.4 Attacks and Defenses 70

5.2 Implementation 71

5.3 Experimental Results 73

CHAPTER 6 SILVER LINING: ENFORCING SECURE INFORMATION FLOW AT THE CLOUD EDGE 77

6.1 System Overview 80

6.1.1 Cloud Structure 80

6.1.2 Access and Information Flow Control 82

6.1.3 Threat Model 82

6.1.4 MapReduce Paradigm 84

6.1.5 Policy Specifications and Flow Tracking 84

6.1.6 In-lined Reference Monitor Design 86

6.1.7 Architecture and Protocol 88

6.2 Implementation 89

6.3 Experimental Results 92

CHAPTER 7 RELATED WORK 95

7.1 Hatman 95

7.2 AnonymousCloud 97

7.3 Penny 100

7.4 CloudCover 103
7.5 SilverLine 105
CHAPTER 8 CONCLUSION 108
8.1 Hatman 108
8.2 AnonymousCloud 109
8.3 Penny 110
8.4 CloudCover 111
8.5 SilverLine 112
REFERENCES 115
VITA

LIST OF FIGURES

2.1 A Hatman job replicated k times and distributed across n data nodes per replica group. 11

2.2 Success rate over time 18

2.3 Two-frame comparison for $n = 2$ 18

2.4 k versus success rate 19

2.5 k versus activity time 19

3.1 System architecture of AnonymousCloud 23

3.2 Authentication and circuit construction message sequence. Solid lines denote direct communications, whereas dashed lines denote anonymous communication through the Tor circuit. 26

3.3 Privacy enforcement success as a function of Tor circuit length k in a cloud of $p = 30\%$ malicious slave nodes 28

3.4 Privacy enforcement success for Tor circuits of length $k = 3$ as a function of percentage p of malicious slave nodes 29

3.5 Communication overhead for different circuit lengths k 30

4.1 Penny message propagation 37

4.2 Agent join operation 38

4.3 Agent leave operation 40

4.4 Publish protocol for traditional file objects 41

4.5 Request protocol for traditional file objects 42

4.6 Performance comparisons for $c = 2k$ 48

4.7 Performance comparisons for $c = 3k$ 49

4.8 Negative feedback experiment success rates 51

4.9 Half-correct behavior experiment success rates 52

4.10 Malware propagation experiment success rates 53

4.11 Malware propagation rates 54

4.12 RDF negative feedback experiment success rates 56

4.13 RDF half-correct behavior experiment success rates 57
4.14 RDF malware propagation experiment success rates 58
4.15 RDF malware propagation rates 59
5.1 System architecture of CloudCover 66
5.2 CloudCover checkpointing and validation 67
5.3 SHA-1 computation verification time 74
5.4 MD5 computation verification time 74
5.5 SHA-1 verification times with 6 checkpoints 75
5.6 Original computation runtimes vs. verification runtimes for insertion sort on a 6-node Hadoop MapReduce cluster 76
6.1 Sample Policy File 85
6.2 A sample Information Flow Graph (IFG). Edge (role1, file2) is rejected because its addition would complete disallowed flow (file4, file2). 86
6.3 System architecture of SilverLine 88
6.4 Performance measurement with increasing concurrency 92
6.5 Performance measurement with increasing IFG size 93
6.6 Comparing performance of different MapReduce jobs 94

LIST OF TABLES

2.1 Hatman code breakdown 16

CHAPTER 1

INTRODUCTION

Revolutionary advances in hardware, networking, middleware, and virtual machine technologies have led to an emergence of new, globally distributed computing platforms, namely cloud computing, that provide computation facilities and storage as services accessible from anywhere via the Internet without significant investments in new infrastructure, training, or software licensing. Infograph reports that 63% of financial services, 62% of manufacturing, 59% of healthcare, and 51% of transportation industries are using cloud computing services (Meijer, 2012). According to Rackspace (Nicholson et al., 2013), this pay-as-you-go service saves around 58% of cost.

As a result, more than 50% of global 1000 companies are projected to store sensitive data in public clouds by 2016 (Smith et al., 2011). However, a significant barrier to the adoption of cloud services is customer fear of data integrity and privacy loss in the cloud (Pearson et al., 2009). A survey by Fujitsu Research Institute reveals that 88% of prospective customers are worried about who has access to their data in the cloud and demand more trustworthiness (Fujitsu, 2010). Additionally, there are mission-critical clouds: for example, the NSA is using MapReduce (Dean and Ghemawat, 2008) clouds for intelligence data mining (Iannotta, 2011), and the NIH is using AWS cloud (Amazon, 2013) for health data management (Cravedi and Randall, 2012). Therefore, cloud computing security has established its paramount importance.

There are numerous security issues involved in clouds, some of which include:

- **Privacy Preservation**: preserving privacy of data and its owners,

- **Computation Integrity**: ensuring computations are correct,

- **Secure Storage**: storing data securely (e.g., via encryption),
- **Authentication and Authorization**: cloud user access control, and
- **Secure Remote Platform Attestation**: detecting and protecting against software tampering.

This dissertation examines most of these issues (all but secure storage) from the perspective of *trust decentralization, minimization, and management* in clouds.

Since users lack full control over resources in clouds, they must rely on trust mechanisms. A dictionary definition of *trust* is, "firm belief in the reliability, truth, ability, or strength of someone or something" (Soanes and Stevenson, 2006). Thus it revolves around assurance and confidence that people, data, entities, information or processes will function or behave in expected ways (Pearson and Benameur, 2010). In a heterogeneous environment, this notion of trust is inevitably difficult to precisely quantify, so there is no universally accepted definition of trust in cloud computing. However, by reducing, eliminating, and/or distributing trust relationships between cloud infrastructure components and users, one can make relative, incremental improvements to the trustworthiness of clouds, improving their security. This relative notion of trust is well established in the general security literature (cf., Blaze et al., 1996, 2009).

The current implementations of clouds (Amazon, 2013; Microsoft, 2013; Apache, 2013) typically have centralized, universal trust of all the cloud nodes. This security paradigm suffers from a major drawback: though the nodes may be considered trustworthy by the clouds, if the attackers can compromise some, or even one, of the nodes in the cloud over time, the whole computation is compromised or data integrity and privacy can be breached. Therefore, this dissertation focuses on how to decentralize these trust relationships in clouds in order to improve security without impairing efficiency.

To deal with this centralized trust in clouds, one can bring forth different possible solutions, which we can categorize into mainly two lines of defense:

- **First Line of Defense**: Most cloud defense technologies seek to prevent attackers from compromising any cloud resources in the first place. One major category of such technologies is *virtualization*, which uses secure operating systems, hardware, and virtual machines to place layers of security between security-sensitive cloud resources and untrusted user activities. However, inevitably these defenses are not perfect. It is prudent to expect that some attackers will penetrate this first line of defense, motivating a second line of defense.

- **Second Line of Defense**: Beneath the first line of defense, one can add a second line of defense to detect and mitigate successful intrusions. The classic approach adopts distributed fault tolerance—for instance, Byzantine fault tolerance. However, many fault tolerant approaches only target adversaries that act purely randomly (e.g., a hostile environment that randomly corrupts computations). In contrast, attackers are typically non-random. They strategically exploit attack vectors that subvert the defense with high probability. This has motivated research on trust management models that seek to harden distributed and decentralized (e.g., peer-to-peer) networks against malicious adversaries. This latter approach is the subject of this dissertation.

We explore the problem of trust management in clouds through five works, described below.

Hatman. Our first proposed work, *Hatman* (Khan and Hamlen, 2012b), decentralizes trust through replication in the cloud, and ensures computation integrity. To evaluate reputation-based trust management in a realistic cloud environment, we augment a full-scale, production-level data processing cloud—Hadoop MapReduce (Apache, 2013; Dean and Ghemawat, 2008)—with a reputation-based trust management implementation based on EigenTrust (Kamvar et al., 2003). The augmented system replicates Hadoop jobs and sub-jobs across the untrusted cloud nodes, comparing node responses for consistency. Consistencies and inconsistencies constitute feedback in the form of agreements and disagreements between nodes. These form a trust matrix whose eigenvector encodes the global reputations of all nodes in the cloud. The global trust vector is consulted when choosing between differing replica responses, with the most reliable response delivered to the user as the job outcome. To achieve high scalability and low overhead, we show that job replication, result consistency checking, and trust management can all be formulated as highly parallelized MapReduce computations. Thus, the security offered by the cloud scales with its computational power.

Different cloud computing platforms may vary in the details of their internal architectures, usually with one or few centralized master nodes and a large collection (e.g., hundreds of thousands) of slave nodes in Hadoop. The trust management system is centralized in the sense that master nodes maintain a small, trusted store of trust and reputation information; however, all computation is decentralized in that trust matrix computations and user-submitted job code is all dispatched to slave nodes.

AnonymousCloud. Our next paradigm, *AnonymousCloud* (Khan and Hamlen, 2012a), decentralizes trust by decoupling billing information from submitted jobs. This work concerns the problem of privacy-preserving computation. AnonymousCloud conceals data provenance from cloud nodes that compute over the data, and conceals recipient identities in the form of IP addresses and ownership labels. Anonymization is achieved through the instantiation of a Tor anonymizing circuit (Dingledine et al., 2004) inside the cloud, through which private data and jobs are anonymously supplied by and returned to users. Circuit length is a tunable parameter k, affording a flexible trade-off between the degree of anonymity and the computational overhead of the circuit. To maintain a pay-per-use business model, clouds must inevitably track ownership information at some level for billing and auditing purposes. AnonymousCloud therefore implements a public-key cryptography-based anonymous authentication that disassociates data ownership metadata from the private data it labels.

Penny. The above frameworks have centralized master nodes in clouds that are trusted for integrity, in order to reuse the existing cloud infrastructure. To eliminate that centralized trust, we can adopt a structured peer-to-peer (P2P) topology, which we present in our work *Penny* (Khan and Hamlen, 2013b). All of the master nodes can act as peers, and they distribute jobs and data between them. However, in order to obtain that level of decentralization, we must abandon the existing cloud structure and implement a whole new protocol.

We have designed, implemented, and tested Penny, a P2P networking protocol that extends Chord (Stoica et al., 2001) with secure integrity- and confidentiality-labeling of shared data. Penny uses a distributed reputation management system based on EigenTrust (Kamvar et al., 2003) to securely manage data labels without the introduction of a central authority. The data labels empower requester peers to avoid downloads of low-integrity data, and allow sender peers to deny low-privilege peers access to high-confidentiality data. In addition, sender peers may publish and serve their data anonymously, frustrating attacks that seek to single out and target owners of security-relevant data. We have applied Penny to construct a secure, fully decentralized, data management system for traditional data files as well as Resource Description Framework (RDF) data.

CloudCover. Following the constitution of a trustworthy cloud, we can take advantage of it for computation certification as a service from this cloud. We propose this in *CloudCover* (Khan and Hamlen, 2013a), which decentralizes trust as well—this time on the user side. CloudCover allows untrusted Java computations in an untrusted environment to yield a proof of computation integrity as a side-effect of the computation. The proof can then be validated against the original code and the computation's result to formally verify that the result is correct. Neither the computation nor the proof (nor their origins) are trusted by CloudCover. A (possibly forged) proof either proves that a given computation results from a given code, or it does not. If the former, the result is correct regardless of where the proof came from; if the latter, the computation, the proof, or both are untrustworthy. Thus, CloudCover can be formalized as *proof-carrying computation*, similar to proof-carrying code (Necula and Lee, 1998). CloudCover proofs have the advantageous quality that the task of verifying them can be parallelized almost arbitrarily even when the original computation is not parallelizable. Thus, they derive maximal benefit from massively parallel architectures, like clouds.

SilverLine. Our last paradigm, *SilverLine* (Khan et al., 2014) is a novel, more modular framework for enforcing mandatory information flow policies on commodity workflow clouds by leveraging Aspect-Oriented Programming (AOP) and In-lined Reference Monitors (IRMs). Unlike traditional system-level approaches, which typically require modifications to the cloud kernel software, OS/hypervisor, or cloud file system layout, SilverLine automatically in-lines secure information flow tracking code into untrusted job binaries as they arrive at the cloud. This facilitates efficient enforcement of a large, flexible class of information flow and mandatory access control policies without any customization of the cloud or its underlying infrastructure. The cloud and the enforcement framework can therefore be maintained completely separately and orthogonally. We implement and deploy SilverLine on a fullscale, real-world data processing cloud—Hadoop MapReduce.

The remainder of the dissertation proceeds as follows. Chapters 2–6 present each security paradigm described above, respectively. Relevant related work is discussed in Chapter 7. Finally, Chapter 8 concludes with a summary and a discussion of directions for future work.

CHAPTER 2

HATMAN: INTRA-CLOUD TRUST MANAGEMENT FOR HADOOP[1]

While cloud data privacy has received a great deal of popular attention in the literature (cf., Ryan, 2011; Chen and Zhao, 2012), computation integrity also remains a significant problem for large-scale, production-level cloud architectures. An attacker who is able to compromise even one cloud node potentially gains the ability to corrupt the outcomes of all computations allocated to that node. Since clouds subdivide and distribute their computations as widely as possible across their nodes to achieve high performance and scalability, this means that a single compromised node can corrupt the integrity of many or even all of the jobs undertaken by the cloud.

As an example, consider a Hadoop MapReduce cloud (Dean and Ghemawat, 2008) that performs military intelligence data mining, similar to the one currently under development by the U.S. National Security Agency (Iannotta, 2011). An attacker who has compromised just one node in such a cloud can introduce nearly arbitrary error into simple computations, such as word counting or clustering computations, by simply forcing the compromised node to yield false, outlying answers to queries. These answers are summed or averaged into the answers returned by the other nodes, resulting in a final answer (delivered to the user) that is largely dictated by the attacker. Such computational integrity corruption could be applied to frustrate military intelligence data-mining efforts by masking important data correlations or introducing false ones.

[1]© 2012 IEEE. Reprinted, with permission, from Safwan M. Khan and Kevin W. Hamlen. Hatman: Intra-cloud Trust Management for Hadoop. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 494–501, June 2012.

For this reason, a large body of work on cloud security focuses on protecting nodes from being compromised in the first place (cf., Subashini and Kavitha, 2011; Hamlen et al., 2010). Data processing clouds (Du et al., 2009), including Hadoop, execute untrusted, user-submitted code on trusted cloud nodes during job processing, and must therefore remain vigilant against malicious mobile code attacks. Virtualization technologies, including trusted hardware, hypervisors, secure OSes, and trusted VMs are the typical means by which such mobile code is secured (e.g., Berger et al., 2008; Ibrahim et al., 2011). However, a variety of studies have shown that clouds introduce significant new security challenges that make mobile code security a non-trivial, ongoing battle (Christodorescu et al., 2009; Chen et al., 2010; Subashini and Kavitha, 2011; Morsy et al., 2010). For example, the Cloud Security Alliance has identified insecure cloud APIs, malicious insiders, shared technology issues, service hijacking, and unknown risk profiles all as top security threats to clouds (Cloud Security Alliance, 2010).

We therefore examine trust management as a second line of defense for cloud computation integrity enforcement. Trust management systems (Blaze et al., 1996) weather (rather than preclude) malicious behavior in distributed systems by tracking reputations of untrusted agents (e.g., cloud nodes) over time. Agents who frequently exhibit behavior characterized as malicious by more trustworthy agents accrue poor reputations, and therefore become distrusted by the rest of the system. This facilitates detection and rejection of misbehaving agents without the need to modify the underlying hardware, software, or communication protocols of each agent in the system.

To evaluate reputation-based trust management in a realistic cloud environment, we augment a full-scale, production-level data processing cloud—Hadoop MapReduce (Apache, 2013; Dean and Ghemawat, 2008)—with a reputation-based trust management implementation based on EigenTrust (Kamvar et al., 2003). The augmented system replicates Hadoop jobs and sub-jobs across the untrusted cloud nodes, comparing node responses for consistency. Consistencies and inconsistencies constitute feedback in the form of agreements and disagreements between nodes. These form a trust matrix whose eigenvector encodes the global reputations of all nodes in the cloud. The global trust vector is consulted when choosing between differing replica responses, with the most reliable response delivered to the user as the job outcome.

To achieve high scalability and low overhead, we show that job replication, result consistency checking, and trust management can all be formulated as highly parallelized MapReduce computations. Thus, the security offered by the cloud scales with its computational power. Our primary contributions are therefore as follows:

- We implement and evaluate intra-cloud trust management for a real-world cloud architecture—Hadoop.
- Our system adopts a data-centric approach that recognizes job replica disagreements (rather than merely node downtimes or denial of service) as malicious.
- We show how MapReduce-style distributed computing can be leveraged to achieve purely passive, full-time, yet scalable attestation and reputation-tracking in the cloud.

Consequently, Hatman decentralizes trust through replication in the clouds and ensures computation integrity.

2.1 System Overview

2.1.1 Overview of Hadoop Architecture

The Hadoop Distributed File System (HDFS) (Apache, 2013) is a master/slave architecture designed to run on commodity hardware. Each HDFS cluster has a single *NameNode* master, which manages the file system namespace and regulates access to files by customers. In addition, there are a number of *DataNodes*, usually one per node in the cluster, which manage storage attached to the nodes on which they run. The DataNodes are arranged in racks for replication purposes. Customers communicate with the NameNode, which coordinates the services from the DataNodes.

MapReduce (Dean and Ghemawat, 2008) is an increasingly popular distributed programming paradigm used in cloud computing environments. It expedites the processing of large datasets using inexpensive cluster computers. Additional advantages include load balancing and fault tolerance.

In this research work we use Hadoop's MapReduce framework (Apache, 2013). In Hadoop, the unit of computation is called a *job*. Customers submit jobs to Hadoop's JobTracker component. Each job has two phases: Map and Reduce. The Map phase maps input key-value pairs to a set of intermediate key-value pairs. The Reduce phase reduces the set of intermediate key-value pairs that share a key to a smaller set of key-value pairs traversable by an iterator. When a job is submitted to the JobTracker, Hadoop attempts to place the Map processes near to the input data in the cluster to reduce the communication cost. Each Map process and Reduce process works independently without communication.

2.1.2 Hatman Architecture

Hatman (HAdoop Trust MANager) augments Hadoop NameNodes with reputation-based trust management of their slave DataNodes. The trust management system is centralized in the sense that NameNodes maintain a small, trusted store of trust and reputation information; however, all computation is decentralized in that trust matrix computations and user-submitted job code is all dispatched to DataNodes. NameNode computations therefore remain restricted to simple bookkeeping operations related to job dispatch. This keeps the system scalable and maintains high trustworthiness of NameNodes by minimizing their attack surfaces.

Hatman users submit Hadoop jobs J with two additional parameters: (1) a *group size* n and (2) a *replication factor* k. The NameNode distributes user-submitted computation

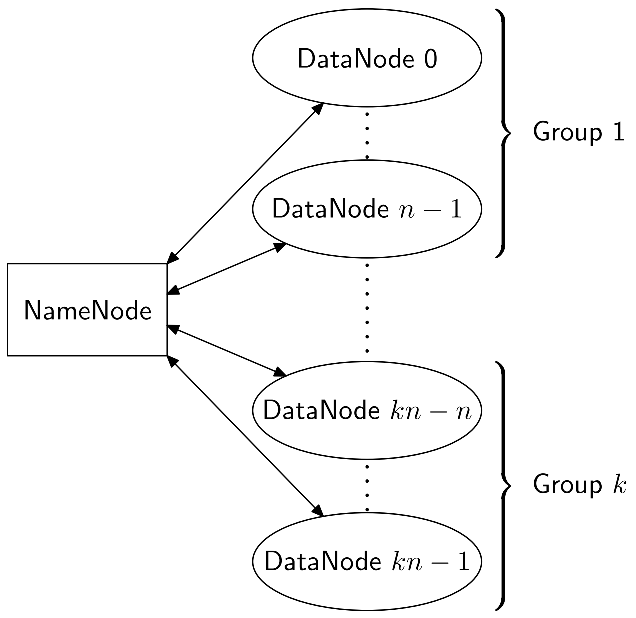


Figure 2.1. A Hatman job replicated k times and distributed across n data nodes per replica group.

J across kn DataNodes, as illustrated in Figure 2.1. Each group of n nodes independently processes job J, with any sub-jobs being redistributed to the nodes of the group that spawned it. Different groups are permitted to have some common members (though this is unlikely when kn is small relative to the size of the cloud), but no two groups are identical. Increasing n therefore yields higher parallelism and increased performance, whereas increasing k yields higher replication and increased security.

Interpretation of the results of these replicated computations proceeds according to Algorithm 1. Line 3 first collects candidate results from each of the k replica groups. The collected results are compared pairwise by lines 7 and 9. Hadoop job results are simply files, which can be partitioned and compared in a highly distributed fashion; therefore, comparison of non-trivial results is implemented by line 9 as a second Hatman job over freshly selected nodes and groups. The comparison job recursively leverages the trust management system to ensure high data integrity, resulting in a reliable comparison.

Algorithm 1 Hatman job processing

```
Input: job $J$, group size $n$, replication factor $k$
Output: job result $r$
 1: Choose $k$ unique groups $G_g$ each of size $n$
 2: for all groups $G_g$ do
 3:   $r_g \leftarrow \mathit{HadoopDispatch}(G_g, J)$
 4: end for
 5: for all pairs $(G_g, G_h)$ with $g \neq h$ do
 6:   if $r_g$ and $r_h$ are small then
 7:     $eq \leftarrow (r_g =? \, r_h)$
 8:   else
 9:     $eq \leftarrow \mathit{HatmanDispatch}(r_g =? \, r_h)$
10:   end if
11:   for all $(i, j) \in G_g \times G_h$ with $i \neq j$ do
12:     $C_{ij} \leftarrow C_{ij} + 1$ (and $C_{ji} = C_{ij}$)
13:     if $eq = \mathit{true}$ then
14:       $A_{ij} \leftarrow A_{ij} + 1$ (and $A_{ji} = A_{ji}$)
15:     end if
16:   end for
17: end for
18: if time to update trust vector then
19:   $T \leftarrow \mathit{HatmanDispatch}(\mathit{tmatrix}(A, C))$
20:   $t \leftarrow \mathit{HatmanDispatch}(\mathit{EigenTrust}(T))$
21: end if
22: $m \leftarrow \arg\max_g \mathit{eval}(G_g)$
23: return $r_m$
```

Lines 11–16 tally agreements and disagreements between the groups in a local trust matrix. Lines 18–21 periodically use this to compute a global trust vector for all nodes in the system. Finally, line 22 uses the global trust vector to evaluate the most trustworthy result to return to the user. We next discuss the details of the local trust matrix, the global trust matrix computation, and the evaluation function, respectively.

A large category of Hadoop jobs tend to be stateless and deterministic (Gedik et al., 2008; Du et al., 2011, 2010). Thus, when all nodes are reliable, all k replica groups yield identical results r_g with high probability. However, when some nodes are malicious or unreliable, the NameNode must choose which of the differing responses to deliver to the user, and it must decide how the reputations of members of disagreeing groups are affected by the disagreement.

To make this decision, we appeal to a trust model defined by a local trust matrix $T_{ij} = \alpha_{ij} t_{ij}$, where $t_{ij} \in [0, 1]$ measures how much agent i trusts agent j, and $\alpha_{ij} \in [0, 1]$ measures agent i's relative confidence in his choice of t_{ij} (Jakubowski et al., 2009). The confidence values are relative in the sense that $\sum_{i=1}^{N} \alpha_{ij} = 1$, where N is the total number of agents.

In Hatman, DataNode i trusts DataNode j proportional to the percentage of jobs shared by i and j on which i's group agreed with j's group. That is, $t_{ij} = A_{ij}/C_{ij}$ where C_{ij} is the number of jobs shared by i and j and A_{ij} is the number of those jobs on which their groups' answers agreed. DataNode i's relative confidence is the percentage of assessments of j that have been voiced by i:

$$\alpha_{ij} = \frac{C_{ij}}{\sum_{k=1}^{N} C_{kj}} \tag{2.1}$$

Multiplying $T_{ij} = \alpha_{ij} t_{ij}$ therefore yields matrix

$$T_{ij} = \frac{A_{ij}}{\sum_{k=1}^{N} C_{kj}} \tag{2.2}$$

This is the computation performed by $\textit{tmatrix}(A, C)$ in line 19 of Algorithm 1.

This formula is well-defined whenever j has shared at least one job with another DataNode (making the denominator non-zero). When j has not yet received any shared jobs, we allow all DataNodes to initially trust j (so $t_{ij} = 1$) with uniform confidence ($\alpha_{ij} = 1/N$).

This differs from EigenTrust (Kamvar et al., 2003), which initially distrusts new agents because it targets networks with potentially uncontrolled churn. A default distrust of new peers disincentivizes leaving and rejoining such a network to reset reputation. In contrast, clouds typically have more controlled churn; new cloud nodes undergo some form of validation and authorization by organization personnel at installation and cannot leave and rejoin the network arbitrarily. Therefore, Hatman trusts new nodes by default and reduces that trust in response to evidence of compromise.

Following EigenTrust, line 20 computes the left eigenvector of local trust matrix T to obtain a vector t of global reputations for all DataNodes. Once again, this computation is formulated as a distributed Hatman job across a fresh set of nodes and replica groups. The trust vector is recomputed at regular intervals and at idle periods rather than after every job to avoid overburdening the network.

Reputation vector t is used as a basis for evaluating the trustworthiness of each group's response. We employ evaluation function

$$eval(G) = w\frac{|G|}{|S|} + (1-w)\frac{\sum_{i \in G} t_i}{\sum_{i \in S} t_i} \tag{2.3}$$

where $S = \cup_{j=1}^{k} G_j$ is the complete set of DataNodes involved in the activity, and weight $w \in [0, 1]$ defines the relative importance of group size versus group collective reputation in assessing trustworthiness. In Section 2.3 we observe highest accuracy with $w = 0.2$, demonstrating that reputation is about 4 times more effective than simple majority voting for identifying integrity violations. The result yielded by the group with the highest evaluation score is the one returned to the user.

2.1.3 Activity Types

An *activity* is a tree of sub-jobs whose root is a job J submitted to Algorithm 1. There are three different types of activities that Hatman undertakes: *user-submitted activities*, *bookkeeping activities*, and *police activities*.

User-submitted activities are jobs submitted by cloud customers. These receive highest priority in the system, with parameters n and k chosen by the user (and perhaps entailing higher customer cost in response to demands for greater parallelism and replication).

Bookkeeping activities are result-comparison and trust matrix computation jobs submitted by lines 9, 19, and 20 of Algorithm 1. These inherit the priority of the user-submitted job with which they are associated, and receive high replication factors k to ensure their integrity.

Police activities are dummy jobs (e.g., replayed user-submitted activities or stock jobs) whose sole purpose is to exercise the system. These are undertaken during periods of low load to help trust matrix T converge more quickly. The results of police activities are discarded, providing a safe means to assess reliability of low-reputation nodes without risking delivery of low-integrity results to users.

2.1.4 Attacker Model and Assumptions

We assume that attackers can compromise DataNodes but not NameNodes. NameNodes do not execute any user-submitted code, and have a substantially simpler computing architecture relative to DataNodes, reducing their vulnerability to attack. We also assume that communication between NameNodes and DataNodes is cryptograhpically protected, so that a man-in-the-middle cannot forge or replay messages.

Following prior work (Du et al., 2011, 2010), we assume that most (but not necessarily all) jobs are deterministic and stateless, so that inconsistencies are indicative of integrity violations. This assumption is valid for a large category of data processing jobs that dominate Hadoop and similar cloud frameworks. Non-determinism based on random number generation can be adapted to our system by requiring job-authors to expose random number generator seeds as job inputs, so that they can be duplicated across replica groups.

Attacker-compromised nodes in our model occasionally (or always) submit incorrect results for jobs that they process. Confidentiality and denial of service attacks are outside our scope, but are addressed by a large body of other work (cf., Ryan, 2011; Chen and Zhao, 2012).

Table 2.1. Hatman code breakdown

Component	Description	Lines
ActivityGen	Generate police activities.	300
NameNode Additions:		
TrustMatrix	Compute local trust matrix and dispatch eigenvector job computations.	3500
Evaluator	Evaluate group responses and select results.	4000
Interface	Initialize and finalize jobs.	1500
Job Code:		
Compare	Test results for equivalence.	600
EigenVector	Compute left eigenvector of local trust matrix.	300
Clustering	Police activity code.	600
Total		11000

2.2 Implementation

Our implementation of Hatman consists of about 11,000 lines of Java code added to the open source release of Hadoop v0.20.3. Table 2.1 reports a size breakdown of each component's programming.

The majority of the implementation modifies Hadoop's JobTracker and NetworkTopology modules to adjust the distribution of input and output files, and modify the scheduling of jobs during Map and Reduce phases in accordance with Algorithm 1. This accounts for about 82% of the implementation.

Approximately 1500 lines of additional MapReduce code implement distributed algorithms for result comparison, eigenvector computation, and police activity jobs. For our police activities we used a K-means clustering algorithm that partitions randomly generated data sets of 10,000 data points into 2 clusters. A separate ActivityGen module submits police activities to NameNodes during idle times or other periods of low activity. To maximize the effectiveness of the police jobs, they are submitted with parameters $n = 1$ and $k = 3$. Group size $n = 1$ helps the trust manager reliably trace inconsistencies to one misbehaving node per group, and small, odd replication factor k helps break potential ties with low overhead.

Our test architecture is a Hadoop cluster consisting of 8 DataNodes and 1 NameNode. Node hardware consists of Intel Pentium IV 2.40–3.00GHz processors with 2–4GB memory each, running Ubuntu operating systems. In each test, 2 of the 8 nodes (25%) are malicious, randomly returning correct or incorrect results for each job they are assigned.

2.3 Results and Analysis

In all experiments we used Equation 2.3 for evaluation with group size weighted at $w = 0.2$ and group reputation weighted at $1 - w = 0.8$. This yielded the best success rates in all cases. Police activities were submitted at regular intervals between user-submitted jobs, and account for 30% of the network's overall load.

Figure 2.2 illustrates Hatman's success rate in selecting correct job outputs in a Hadoop cloud of 25% malicious nodes, with user-submitted jobs having group size $n = 1$ and replication factor $k = 3$. Each data point reports the average success rate over a frame consisting of 20 user activities. Initially the success rate is 80% because there is initially no reputation information for the nodes. However, by frame 8 all the malicious nodes have been identified and the success rate rises to 100%. The average success rate over all frames is 89%.

Figure 2.3 considers the same experiment but with the results divided into only two frames (1st half and 2nd half) of 100 activities each, an increased group size of $n = 2$, and an increased replication factor k ranging from 3 to 7. The plot for the 2nd half of the experiment is substantially above the one for the 1st half in all cases, illustrating that after 100 activities the trust algorithm has converged to 96.33% accuracy on average. As expected, higher replication factors push the success rate even higher—near 100% with $k = 7$.

Figures 2.4–2.5 examine the impact of replication factor k and group size n more directly. Figure 2.4 shows that increasing the replication factor can substantially increase the success

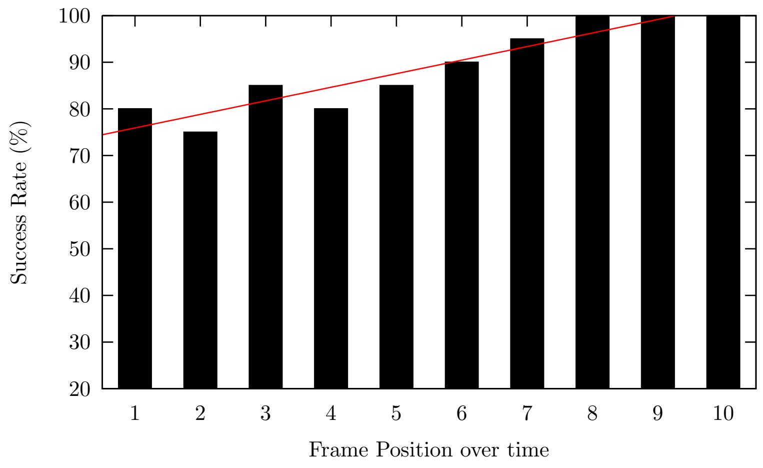


Figure 2.2. Success rate over time

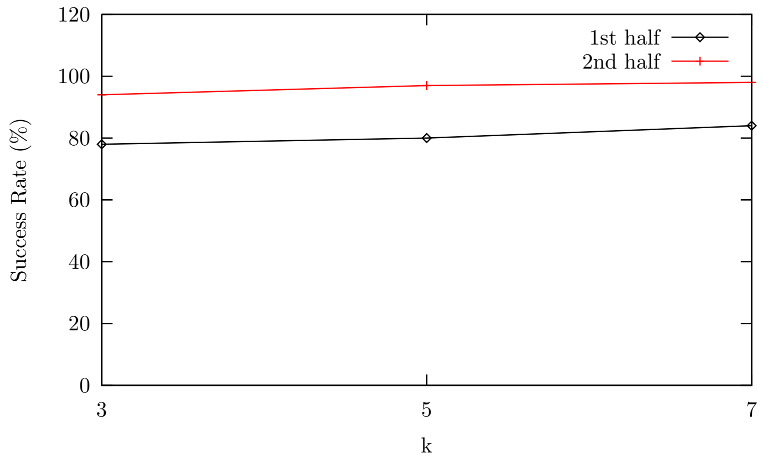


Figure 2.3. Two-frame comparison for $n = 2$

rate for any given frame on average. The impact is more pronounced when n is small because feedback from replica inconsistencies is more node-specific with smaller group sizes, leading to more accurate blame assigned by the trust manager. In fact, with sufficiently high k and low n, accuracy can be pushed almost arbirarily high, as seen by the 100% success rate at $k = 7$ and $n = 1$.

Figure 2.5 demonstrates the high scalability of our approach by showing how activity times remain almost completely flat as k increases. This is because all significant computations associated with replica management are fully parallelized across the entire cloud. Note

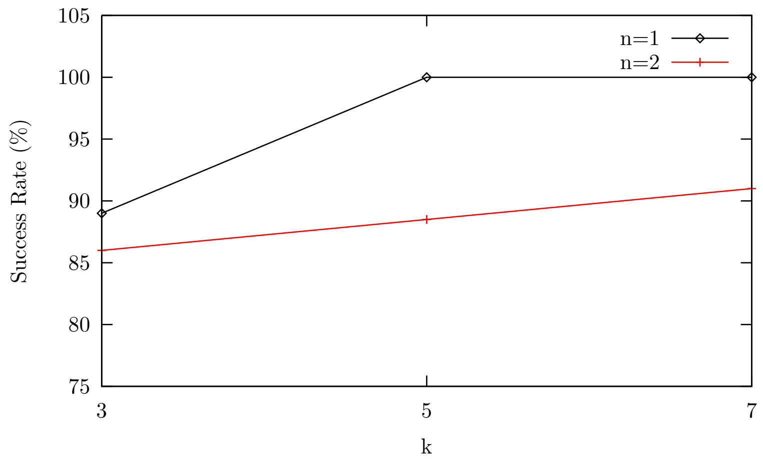


Figure 2.4. k versus success rate

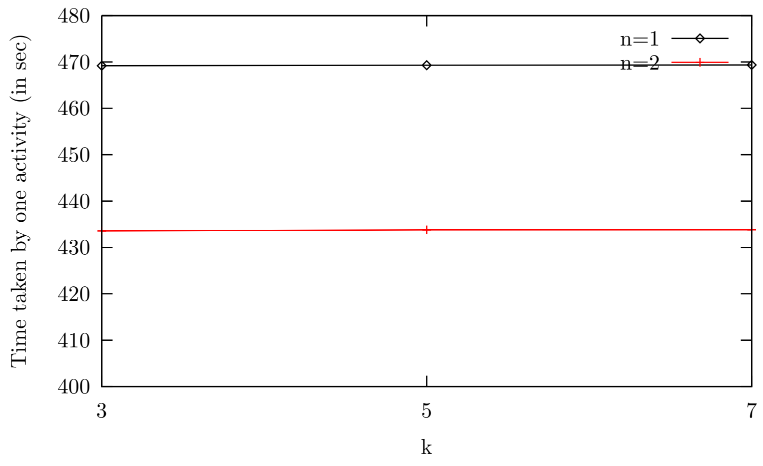


Figure 2.5. k versus activity time

that $k = 7$ and $n = 2$ results in a total load $kn = 14$ that is almost twice the size of our cloud. Nevertheless, activity time remains comparable to $k = 3$ and $n = 2$.

Based on this preliminary evidence, we believe that Hatman will scale extremely well to larger Hadoop clusters with larger numbers of data nodes. Each additional data node adds to the size of the trust matrix, but since all trust matrix operations are distributed across all available data nodes, each additional node also increases the power of the cloud to manage the larger trust computations. This agrees with experimental evidence from prior work showing that EigenTrust and similar distributed reputation-management systems scale well to large networks (West et al., 2010).

CHAPTER 3

ANONYMOUSCLOUD: A DATA OWNERSHIP PRIVACY PROVIDER FRAMEWORK IN CLOUD COMPUTING[1]

This work concerns the problem of privacy-preserving computation in the cloud. The complementary problem of secure storage of private cloud data has been studied extensively in the literature (cf., Ryan, 2011; Chen and Zhao, 2012), but cannot usually be applied while the data is in decrypted form for the duration of a computation. Secure multiparty computation (Lindell and Pinkas, 2009) and differential privacy (Roy et al., 2010) are both powerful approaches to privacy-preserving cloud computation on decrypted data, but are inapplicable to many real-world cloud computations. In particular, jobs submitted to the cloud as arbitrary binary code are difficult to automatically reformulate as secure multiparty computations, and high differential privacy sometimes comes at the expense of highly imprecise, noisy results.

In these cases, the level of privacy can sometimes be improved by concealing data ownership, provenance, and/or semantics from the participants in a computation in addition to (or instead of) anonymizing the data itself. For example, a computation that mines medical data might be deemed insecure if cloud nodes receive sequences of numbers labeled "patient temperatures" with owner id "Mercy Hospital"; however, the same computation might be deemed suitably private if each node receives only unlabeled sequences of numbers amidst a context of millions of other similar anonymous jobs for thousands of diverse, anonymous users.

[1]© 2012 IEEE. Reprinted, with permission, from Safwan M. Khan and Kevin W. Hamlen. AnonymousCloud: A Data Ownership Privacy Provider Framework in Cloud Computing. In *Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 170–176, June 2012.

Our AnonymousCloud framework therefore conceals data provenance from cloud nodes that compute over the data, and conceals recipient identities in the form of IP addresses and ownership labels. Anonymization is achieved through the instantiation of a Tor anonymizing circuit (Dingledine et al., 2004) inside the cloud, through which private data and jobs are anonymously supplied by and returned to users. Circuit length is a tunable parameter k, affording a flexible trade-off between the degree of anonymity and the computational overhead of the circuit.

To maintain a pay-per-use business model, clouds must inevitably track ownership information at some level for billing and auditing purposes. AnonymousCloud therefore implements a public-key cryptography-based anonymous authentication that disassociates data ownership metadata from the private data it labels. Thus, a separate manager node that does not have access to the private data can bill customers appropriately using the ownership metadata, while computation nodes that have access to the private data but not the metadata can securely carry out the anonymous job. Managers are trusted not to collude with computation nodes to violate privacy, but all other nodes including the master node are potentially malicious.

Our experimental results show via simulation that AnonymousCloud provides data ownership privacy with a high success rate against the collective efforts of a large percentage of attackers in the system, and does so with reasonable computational overhead.

Therefore, AnonymousCloud decentralizes the trust by decoupling billing information from the submitted jobs.

3.1 System Architecture

The system architecture of AnonymousCloud is given in Figure 3.1. It consists of a *cloud provider CP* and a separate *manager M*. Each are discussed respectively below, concluding with a discussion of the communication protocol between the two.

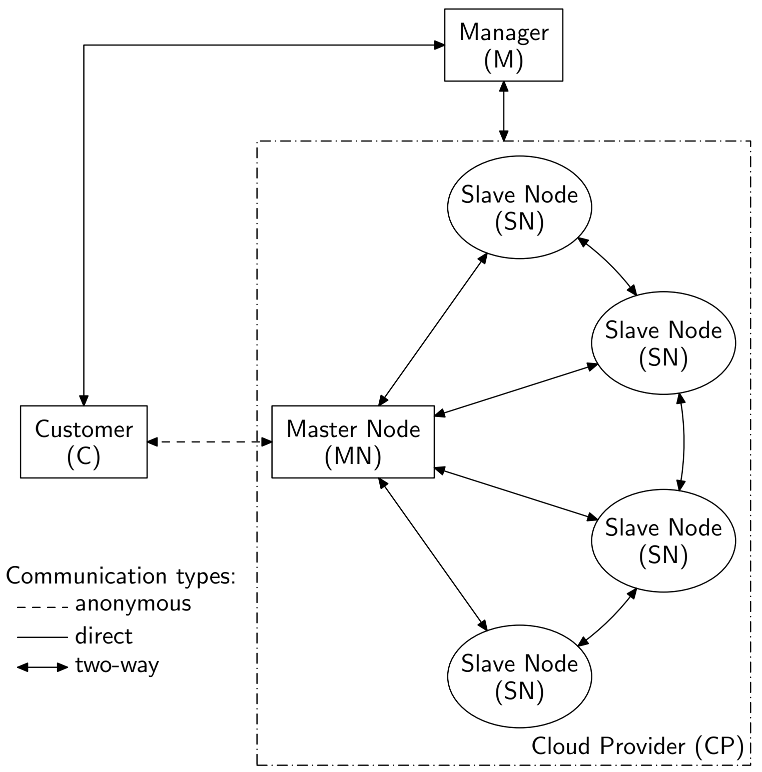


Figure 3.1. System architecture of AnonymousCloud

3.1.1 Cloud Providers

*CP*s provide computation services to customers *C*, who submit computations as *jobs*. Customers can access these services in a pay-as-you-go fashion, with payment managed by the separate manager. Different *CP*s may vary in the details of their internal architectures (cf., Amazon, 2013; Microsoft, 2013; Apache, 2013). We assume only that jobs are submitted to the *CP* via a centralized *master node MN*, which partitions and schedules sub-computations across a large collection (e.g., hundreds of thousands) of *slave nodes SN*s. All *SN*s are therefore directly connected to the *MN*, and there is arbitrary connectivity between the *SN*s.

AnonymousCloud amends Tor functionality (Dingledine et al., 2004) to the *MN* and *SN*s without modifying the job allocation and scheduling details of the cloud in any way. All principals (M, C, MN, and *SN*s) are additionally equipped with public-private key pairs from a well established *certificate authority* CA. The public keys work as the symmetric or mutual keys during Tor circuit construction.

3.1.2 Managers

Managers are separate from the CP's computing infrastructure, and facilitate only customer authentication and billing. They have four primary responsibilities related to our work:

- M provides central storage of public keys for MN and SNs and serves them to C on request.
- M maintains a graph of SN connectivity. This facilitates Tor circuit construction by encoding the universe of available circuit links for circuit initialization.
- M provides each C a unique access token t and credentials c (e.g., a password) via which Cs can authenticate themselves to M to obtain cloud services.
- M additionally generates a unique nonce n for each of C's transactions to protect the authentication system against replay attacks. The authentication protocol is described in greater detail in Section 3.1.3 below.

In deployed implementations, M likely has additional responsibilities related to authentication, such as key revocation, certificate update, auditing, customer billing, etc. These responsibilities are deployment-specific, and are therefore beyond the scope of this work.

3.1.3 Authentication Protocol

The authentication and circuit construction protocol of AnonymousCloud is depicted in Figure 3.2 and detailed in Algorithm 2.

C begins each service transaction by communicating its access token and credentials to M, and requesting an anonymizing circuit of length k. If at least k connected nodes are

Algorithm 2 Authentication and circuit construction protocol

```
1: $C$ asks $M$ to choose $k$ available $SN$s based on $SN$ connectivity
2: if $C$ has invalid token $t$ or invalid credentials $c$ then
3:   $M$ rejects the request from $C$
4: else
5:   repeat
6:     $M$ selects $k$ $SN$s (or the most available)
7:     $M$ provides $C$ with public keys $K_{SN}$ and $K_{MN}$ and fresh nonce $n$
8:     $C$ validates keys $K_{SN}$ and $K_{MN}$ with the $CA$
9:     if any key fails validation by the $CA$ then
10:      $M$ revokes the invalid keys
11:    end if
12:  until all keys are valid
13:  $C$ performs Tor circuit construction (Dingledine et al., 2004) over the SNs using the $K$s as symmetric keys
14:  $C$ signs $t$, $c$, and $n$ with private key $k_C$ and encrypts it with public key $K_M$, yielding $m = \langle t, \langle t, c, n \rangle_{k_C} \rangle_{K_M}$
15:  $C$ sends $\langle m, data \rangle_{K_{MN}}$ in layered encryption format over the circuit to $MN$
16:  $MN$ anonymously receives and decrypts the message with private key $k_{MN}$
17:  $MN$ forwards $m$ to $M$ for authentication
18:  $M$ decrypts $m$ using $k_M$ and verifies signature $k_C$ using $K_C$, yielding $t$, $c$, and $n$
19:  $M$ verifies $t$, $c$, and $n$; and it verifies $K_C$ with the $CA$
20:  if authentication fails then
21:    $M$ returns *false* to $MN$
22:    $MN$ discards the service request
23:  else
24:    $M$ returns *true* to $MN$
25:    $MN$ dispatches the *data* computation
26:    $MN$ anonymously returns the result to $C$ over the circuit
27:  end if
28: end if
```

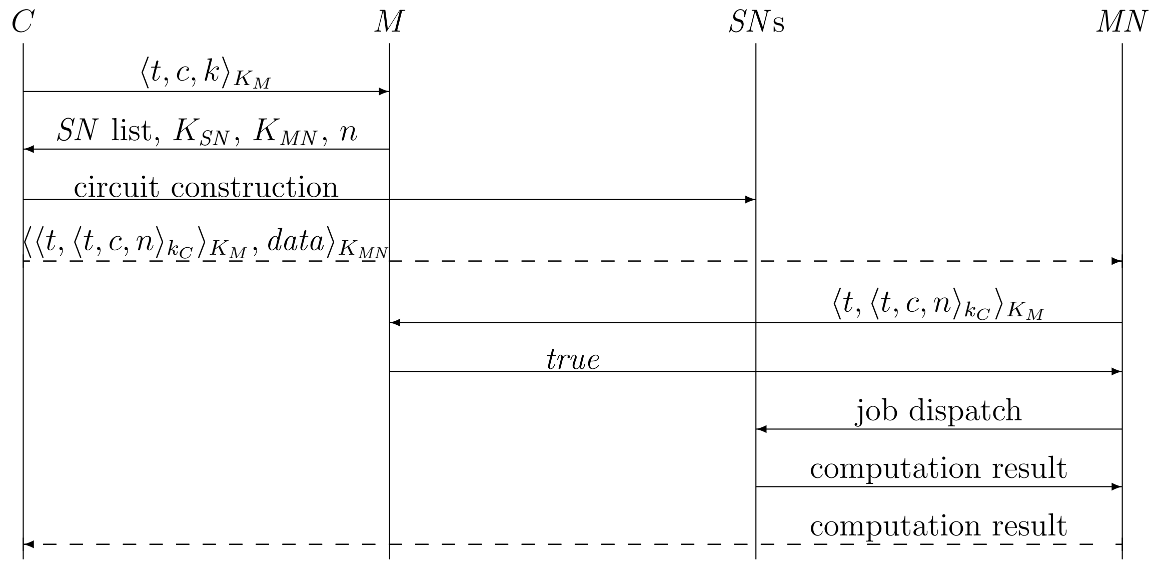

C
M
SNs
MN
$\langle t, c, k\rangle_{K_M}$
SN list, K_{SN}, K_{MN}, n
circuit construction
$\langle\langle t, \langle t, c, n\rangle_{k_C}\rangle_{K_M}, data\rangle_{K_{MN}}$
$\langle t, \langle t, c, n\rangle_{k_C}\rangle_{K_M}$
true
job dispatch
computation result
computation result

Figure 3.2. Authentication and circuit construction message sequence. Solid lines denote direct communications, whereas dashed lines denote anonymous communication through the Tor circuit.

available, M returns such a list; otherwise it may offer a list shorter than k. The returned list includes the public keys K_{SN} of all the selected slave nodes, as well as the public key K_{MN} of the master node. M also generates a fresh nonce n for C and stores a local copy. To prevent replay attacks (Syverson, 1994), the next service request from C will only be authenticated by M if it is labeled with n.

In step 8 of Algorithm 2, C verifies the certificates with the certificate authority and stores them locally. To lessen the load, C may cache these results to avoid re-authenticating certificates that have not changed.

C then transmits the requested computation and its data anonymously via the Tor circuit to MN in step 15. MN can read the data but not the encrypted ownership metadata $m = \langle t, \langle t, c, n\rangle_{k_C}\rangle_{K_M}$. It therefore forwards m to M for validation. M can read metadata m by decrypting it using its private key k_M, however it has no access to the associated job's data. M verifies C's digital signature using public key K_C, and validates K_C's certificate with the certificate authority (possibly caching the results to more efficiently service future requests). The access tokens t inside and outside the digital signature are additionally compared for equality, the credentials c are validated against t, and the nonce n is checked against the local copy. If all these steps succeed, M invalidates the nonce and returns *true* to MN; otherwise it returns *false* and the request is denied.

Upon successful authentication, MN dispatches the requested computation in accordance with the CP's internal architecture and protocols. If customer billing is based on computational resource consumption or other information that only becomes available as the computation progresses, MN can report such information to M without knowing the job's owner by tagging it with encrypted authentication information m. M can then attribute the incurred expenses to the correct customer.

Once the computation is complete, its results are anonymously delivered to C via the Tor circuit. The Tor circuit is then dismantled and its resources reclaimed by the CP.

3.2 Results and Analysis

We implemented AnonymousCloud in a simulation setup using Java, with experiments designed to measure the resilience of the system against privacy attacks and the computational overhead introduced by privacy protections. Each experimental data point is the result of simulating 1000 customer service requests to a cloud consisting of 1 master node and $N = 1000$ slave nodes. The simulation model includes the high-level protocol outlined in Section 3.1.3 but not low-level details of the underlying network and encryption operations, which are expected to be specific to each deployment.

A successful attack against our system is defined as the *linkability* (Pfitzmann and Hansen, 2010) of private data to its corresponding ownership metadata by one or more malicious principals. Principals include the manager, the master node, and all slave nodes. Ownership metadata includes customer pseudonyms (*viz.*, access tokens and IP addresses) and authentication credentials. We assume that private data does not include pseudonyms

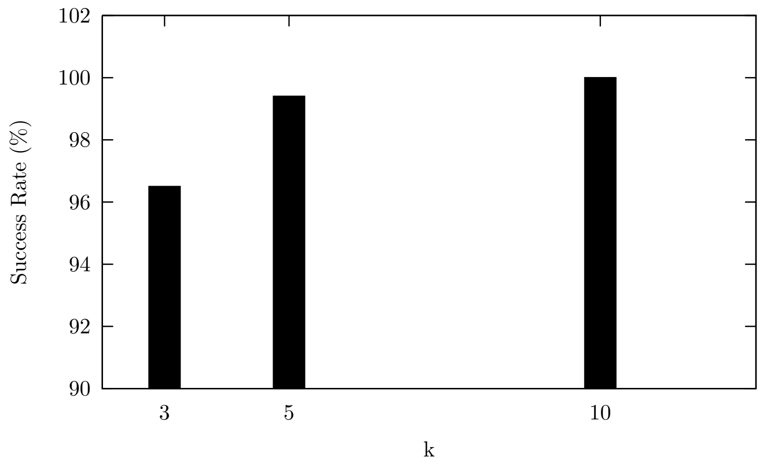


Figure 3.3. Privacy enforcement success as a function of Tor circuit length k in a cloud of $p = 30\%$ malicious slave nodes

or other information from which customer identities can be inferred; anonymizing the private data is the subject of related work.

In order for an attack against AnonymousCloud to succeed, the manager or master node (or both) must be malicious. Managers are the only principals that receive decryptable access tokens or credentials, and all other communications involving pseudonyms and data are conducted via Tor circuits having the master node as the only untrusted endpoint. Managers are separate from CPs and have a much smaller attack surface because they do not process customer-submitted computations. Our experiments therefore assume that managers are trusted, but that master nodes are always malicious. In addition, we assume that a percentage p of slave nodes are also malicious and collude with the malicious master node in an effort to violate privacy.

Figure 3.3 plots the average privacy enforcement success rate for different Tor circuit lengths k in a cloud with a malicious master node and 30% malicious slave nodes. If $k = 0$, AnonymousCloud does not provide any anonymity; furthermore, any length less than 3 significantly increases the ease of successful end-to-end timing attacks (Hopper et al., 2010). We therefore restrict our attention to circuit lengths of at least 3. At $k = 3$ we obtain an

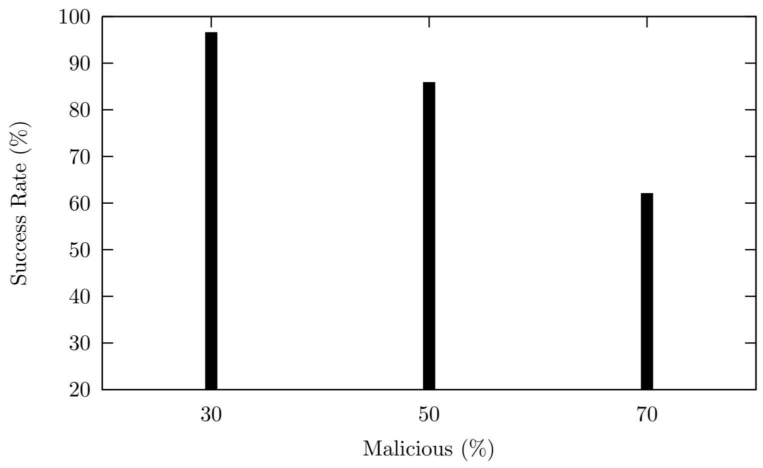


Figure 3.4. Privacy enforcement success for Tor circuits of length $k = 3$ as a function of percentage p of malicious slave nodes

already high success rate of 96.5%. Increasing k to 5 further elevates this 99.4%, and at $k = 10$ there were no privacy failures at all.

Figure 3.4 plots the success rate of a fixed circuit length $k = 3$ in clouds with varying percentages p of malicious slave nodes. The results show how resilient our system is against malicious collectives. Even when clouds are 50% malicious, AnonymousCloud attains an 85.8% privacy preservation rate with just $k = 3$. When 70% of the cloud is malicious, the success rate drops to 62%, indicating that longer circuits are required to resist such pervasive attacks.

The results reported in Figures 3.3 and 3.4 can be generalized by observing that with high probability all k slave nodes in a Tor circuit must collude in order to compromise security. Thus, the curves in Figures 3.3 and 3.4 approximate the formula for random sampling without replacement:

$$\mathit{success} \approx 1 - \binom{pN}{k} \bigg/ \binom{N}{k} \tag{3.1}$$

Privacy inevitably comes at some computational expense. It is therefore important to consider the computational cost associated with the introduction of anonymizing circuits. Figure 3.5 plots the total number of messages per customer service request required to carry

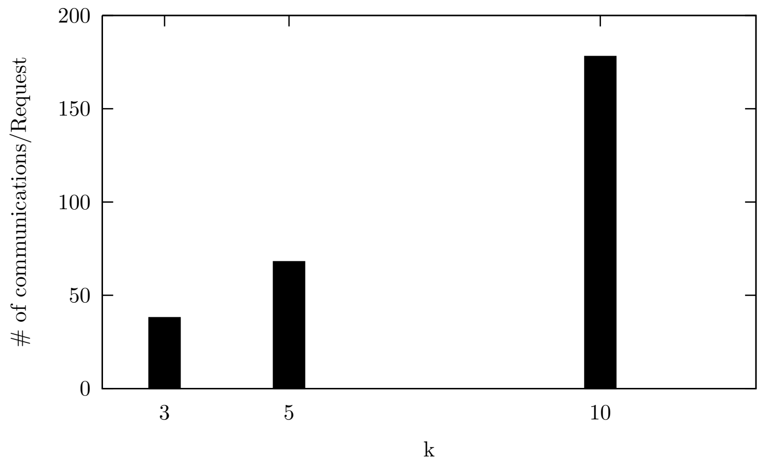


Figure 3.5. Communication overhead for different circuit lengths k

out AnonymousCloud's authentication protocol for varying circuit lengths k. Messages are tallied based on the implementation in the real time. This does not include computational overhead for cryptographic operations, which might noticeably increase the overhead in a real deployment. Although we did not consider these in our simulation, Figure 3.5 nevertheless provides a general picture of the overhead that can be expected. We observe that as circuit length k increases, the total message count per request rises steeply. For $k = 3$ it is 38, and for $k = 5$, it almost doubles to 68. We conclude that there is a significant tradeoff between escalation of k and the overhead cost of communications.

The sharp increase in communications overhead potentially invites denial-of-service attacks by customers who request unreasonably long circuits. We therefore recommend incentivizing reasonable values of k by charging customers proportionally to the communications overhead incurred by their demanded level of privacy. Recall that master nodes can report computational expense information associated with anonymous jobs to managers by labeling it with the encrypted ownership data they received during authentication. This allows the master node to report the expense without knowing the identity of the customer. Managers may also want to impose a mandatory upper limit on k during authentication to further control congestion.

CHAPTER 4

PENNY: SECURE, DECENTRALIZED DATA MANAGEMENT[1]

In the previous two frameworks (Chapter 2 and Chapter 3), we have centralized master nodes in clouds that are trusted for integrity, in order to reuse the existing cloud infrastructure. What if we can change the cloud architecture? Can we get even more decentralization? To answer this question in the affirmative, this chapter adopts a structured peer-to-peer (P2P) toplogy that eliminates centralized trust. All of the master nodes can act as peers and they distribute jobs and data between them. However in order to obtain that level of decentralization, we must abandon the existing cloud structure and implement a whole new protocol.

Peer-to-peer (P2P) networking is a distributed, load-balancing computing paradigm designed to scalably share work loads between peers. Unlike traditional client-server models, each peer in a P2P network is an equally privileged, equipotent participant in the distributed computation or service. This has the advantage of avoiding centralized points of failure that, when successfully attacked, suffice to dismantle the entire network. P2P was first popularized as a vehicle for music-sharing (Napster, 2012), but has since expanded to general-purpose file- and data-sharing applications (e.g., Gnutella, 2010; KaZaA, 2012; BitTorrent, 2012) and is increasingly important as a basis for fault-tolerant cloud computing (Marozzo et al., 2010). Since its inception, it has been tremendously popular and ubiquitous because of its collective computation power, natural load-balancing, and low-cost deployability. For example, it has

[1]© 2013 IJNS. Reprinted, with permission, from Safwan M. Khan and Kevin W. Hamlen. Penny: Secure, Decentralized Data Management. *International Journal of Network Security (IJNS)*, 16(4):289–303, July 2014, forthcoming.

been estimated that BitTorrent traffic accounts for roughly 27–55% of all Internet traffic (depending on geographical location) as of February 2009 (Schulze and Mochalski, 2009).

However, while P2P networks have proven successful for maintaining high data availability under adversarial or noisy conditions, enforcing strong data integrity and confidentiality under these conditions remains a difficult challenge. Integrity enforcement is challenging because P2P networks lack a centralized authority who can identify and evict malicious nodes from the network. Malicious nodes are therefore free to propagate malicious code or untrustworthy data by misrepresenting it as a high-integrity resource available for download (Wang et al., 2010). Approximately 18.5% of all BitTorrent downloads contain malware (Berns and Jung, 2008) as a result. Confidentiality enforcement is impeded by the explicit divulgence of sender peer positions during overlay communications while the requesting peer remains anonymous (Borisov, 2005; Ciaccio, 2006). This allows malicious peers to anonymously identify and target purveyors of security-relevant resources.

To address these deficiencies, we have designed, implemented, and tested Penny, a P2P networking protocol that extends Chord (Stoica et al., 2001) with secure integrity- and confidentiality-labeling of shared data. Penny uses a distributed reputation management system based on EigenTrust (Kamvar et al., 2003) to securely manage data labels without the introduction of a central authority. The data labels empower requester peers to avoid downloads of low-integrity data, and allow sender peers to deny low-privilege peers access to high-confidentiality data. In addition, sender peers may publish and serve their data anonymously, frustrating attacks that seek to single out and target owners of security-relevant data.

We have applied Penny to construct a secure, fully decentralized, data management system for traditional data files as well as Resource Description Framework (RDF) data. RDF is a popular web data format that is of particular importance to Semantic Web technologies. Stores of RDF data can be extremely large and security-sensitive, resulting in an increasing demand for secure distributed computing paradigms that can manage them efficiently (cf., Khaled et al., 2010; Cai et al., 2004). Managing RDF data in a P2P network has the advantage of facilitating dynamic, low-cost growth of the network to accommodate expansion of the data set without sacrificing the security guarantees traditionally associated with more centralized networks, such as clouds. Though there has been much research within the semantic web community on the security and privacy of RDF data, few works consider a fully decentralized approach like the one considered here.

Our prior work proposed Penny's overlay and resource-sharing protocol as a preliminary study without any implementation or experiments (Tsybulnik et al., 2007). We here extend that theoretical work with improvements to the architectural design, new formulas for computing data integrity and confidentiality labels, empirically determined optimal neighborhood sizes, new publish and request protocols adapted for RDF data, and other new empirically tested algorithms necessary for the system. We also describe a full implementation of the Penny client, supporting traditional files and RDF datasets, with experimental results and analysis.

4.1 Penny Network

Penny implements a standard Chord ring, but with an extended form of reputation tracking: For each peer and data object, Penny allots k *score-manager* peers and k *key-holder* peers (respectively) to compute and track the peer or object's trust label(s). Parameter k is fixed at network start and controls the degree of replication; greater k means greater security, since attackers must compromise more peers to successfully corrupt data. Penny strategically positions responsibility-sharing score-managers and key-managers at adjacent ring positions, forming a *neighborhood*. This greatly improves lookup efficiency over standard Chord, since only one overlay message (instead of k) suffices to contact all k replicas. The result is high replication (and therefore high security) with low overhead.

To protect data ownership privacy, data lookups in Penny employ a cryptographically protected extra level of indirection. Data-serving peers first encrypt their requests with the public key of the data item's key-holder, and then ask an arbitrary score-manger to forward the server's key (not its real identifier) and encrypted information to the key-holder. As a result, the key-holder does not know who the real owner of the data item is, and so when someone later requests that data item, the key-holder forwards the request back through the score-manger(s). Meanwhile, the score-mangers do not know which data items are owned by which peers, and thus learn no peer-object associations as they forward the requests. As a result, the ownership information is concealed from all other parties.

We explain this protocol in detail below, beginning with foundational definitions in Section 4.1.1 and proceeding to architectural details in Section 4.1.2.

4.1.1 Definitions

Agents: We refer to the peers in a P2P network as *agents*. Each agent a is assigned an *identifier* id_a by applying a one-way, deterministic hash function to its IP address and port number. We assume that identifiers are unique and that agents cannot influence which identifiers they are assigned. An agent's identifier determines its position in the network's ring structure. When agents are arranged in a ring, each agent has a *predecessor* $pred(a)$ and a *successor* $succ(a)$. We refer to the interval $(id_{pred(a)}, id_a]$ as the *identifier range* of agent a.

Objects and keys: An object o is an atomic item of data (e.g., a file) shared over a P2P network. Each object also has a unique identifier id_o obtained by applying a one-way, deterministic hash function to its name. Objects can be owned by multiple agents. A single *key* is associated with each object and each agent. The keys for object o and agent a are defined by $key_o = h(id_o)$ and $key_a = h(id_a)$ respectively, where h is a one-way, deterministic hash function over the domain of identifiers.

Key-holders and score-managers: Each agent a_1 is assigned a (not necessarily unique) *key-range*, denoted $kr(a_1)$. Agent a_1 is charged with tracking the global integrity and confidentiality labels (discussed later) assigned to all objects o that satisfy $key_o \in kr(a_1)$. In addition, agent a_1 tracks the global trust values (discussed later) assigned to all agents a_2 satisfying $key_{a_2} \in kr(a_1)$. Whenever $key_o \in kr(a_1)$ holds, we refer to agent a_1 as a *key-holder* for object o, and we refer to object o as a *daughter object* of agent a_1. Likewise, whenever $key_{a_2} \in kr(a_1)$ holds we refer to a_1 as a *score-manager* for agent a_2, and we refer to agent a_2 as a *daughter agent* of agent a_1. Every peer in a Penny network acts as both a key-holder for some objects and a score-manager for some peers.

Local confidentiality and integrity labels: Each object o is labeled with a measure of its integrity and confidentiality levels. We denote the integrity and confidentiality labels assigned to object o by agent a as $i_a(o)$ and $c_a(o)$, respectively. Similarly, there are local integrity and confidentiality labels for agents with whom other agents had transactions. Integrity labels measure data quality; confidentiality labels measure who should be permitted to own the data. In Penny, *confidentiality* and *integrity labels* are modeled as real numbers from 0 to 1 inclusive, with 0 denoting lowest confidentiality and integrity and 1 denoting highest confidentiality and integrity.

Local trust values: Trust measures the belief that one agent has that another agent or object will behave as expected or promised. Each ordered pair of agents (a_1, a_2) has a *local trust value* denoted $t_{a_1}(a_2)$ that measures the degree to which agent a_1 trusts agent a_2. Likewise, each ordered pair of agent and object (a, o) has a *local trust value* denoted $t_a(o)$ that measures the degree to which agent a trusts object o. Like confidentiality and integrity labels, trust values range from 0 to 1 inclusive (cf., Kamvar et al., 2003). Local integrity and confidentiality labels are computed and assigned based on local trust values.

Global labels and trust values: Each object o in the system is associated with global integrity and confidentiality labels, denoted i_o and c_o, respectively, and measured by global trust values T_o. Likewise, each agent a is associated with global integrity and confidentiality labels, denoted i_a and c_a, respectively, and measured by global trust values T_a. Key-holders with a common key-range compute T_o and score-managers with a common key-range calculate T_a using Secure EigenTrust (Kamvar et al., 2003). Thus, the global labels and global trust values for any object o and for any agent a can be acquired by any agent in the network by contacting all key-holders a_{kh} for object o, and all score-managers a_{sm} for agent a.

4.1.2 Network Architecture

Identifier Space and Neighborhood

A Penny ring is like a Chord ring, with Penny's identifier ranges being equal to Chord's key-ranges. However, a Penny agent's key-range strictly subsumes its identifier range, and agent key-ranges are not unique. Key-ranges are assigned in a Penny ring so that for every agent a, there are between $\min(k, n)$ and c agents in the ring whose key-ranges are equal to $kr(a)$, where n is the total number of agents and c is a fixed bound on neighborhood size. (The choice of c is discussed in Section 4.2.1; usually $c = 3k$.) Bounding neighborhood size from below by k limits the influence of malicious agents, because each contributes at most $1/k$ of the responses to a secure query. Bounding it from above by c ensures that lookup is not too costly, and it bounds the storage overhead for finger tables.

Message Routing

An agent can contact all score-managers for a particular agent a, or all key-holders for a particular object o, using $O(\log N + k)$ messages. The first $O(\log N)$ messages propagate the message using the Chord algorithm (Stoica et al., 2001) to an agent whose identifier range includes key_a or key_o, who then forwards it directly to the other $O(k)$ agents in

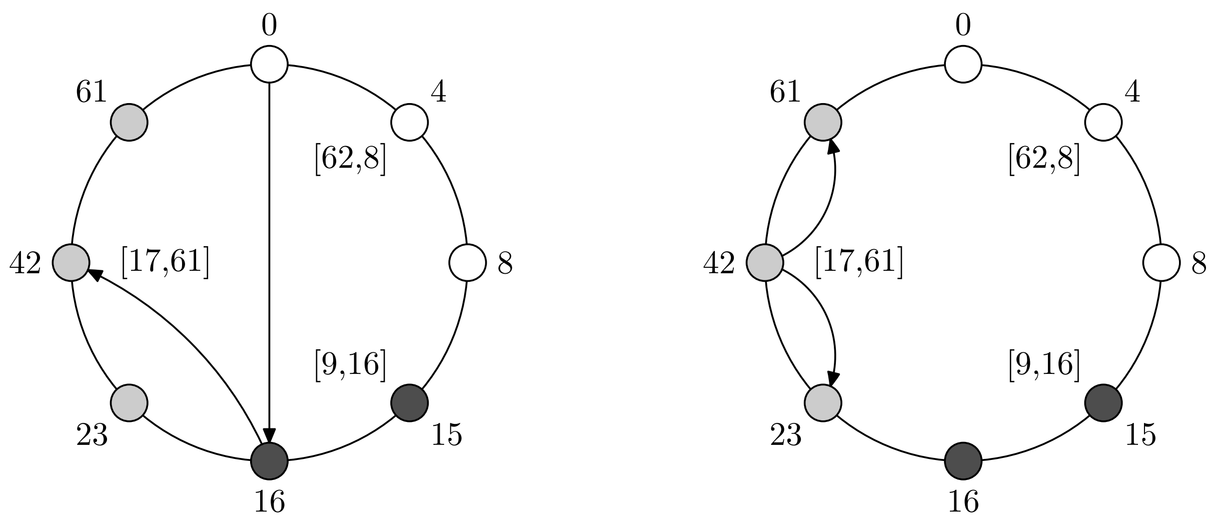


Figure 4.1. Penny message propagation

its neighborhood. Penny therefore reduces the overhead of all network operations that involve contacting key-holders, score-managers, and RDF data owners by a factor of k over EigenTrust. This permits higher replication rates (e.g., $k = 16$) that are often infeasible with past approaches.

As in Chord, each agent a in a Penny ring maintains a finger table that is used to route messages efficiently. For each $i \in [0, m)$, agent a's finger table includes the agent whose identifier range includes $(id_a + 2^i) \bmod 2^m$ (where 2^m is the size of the identifier space). In addition, agent a's finger table also includes an entry for each agent in its neighborhood. The size of each finger table is therefore $O(m + k)$, where k is a constant dictating the number of redundant key-holders assigned to each key.

Figure 4.1 shows an example of the propagation of a Penny message through the resulting ring. In this example, $m = 6$. Agent 0 wishes to send a message to all agents whose key-range includes identifier 28. First, the message is propagated along the ring according to the Chord algorithm to the agent whose identifier range includes 28 (agent 42). This involves first sending the message to the agent whose identifier range includes $0 + 2^4 = 16$ (owner is agent 16), and next to the agent whose identifier range includes $16 + 2^3 = 24$ (owner is agent

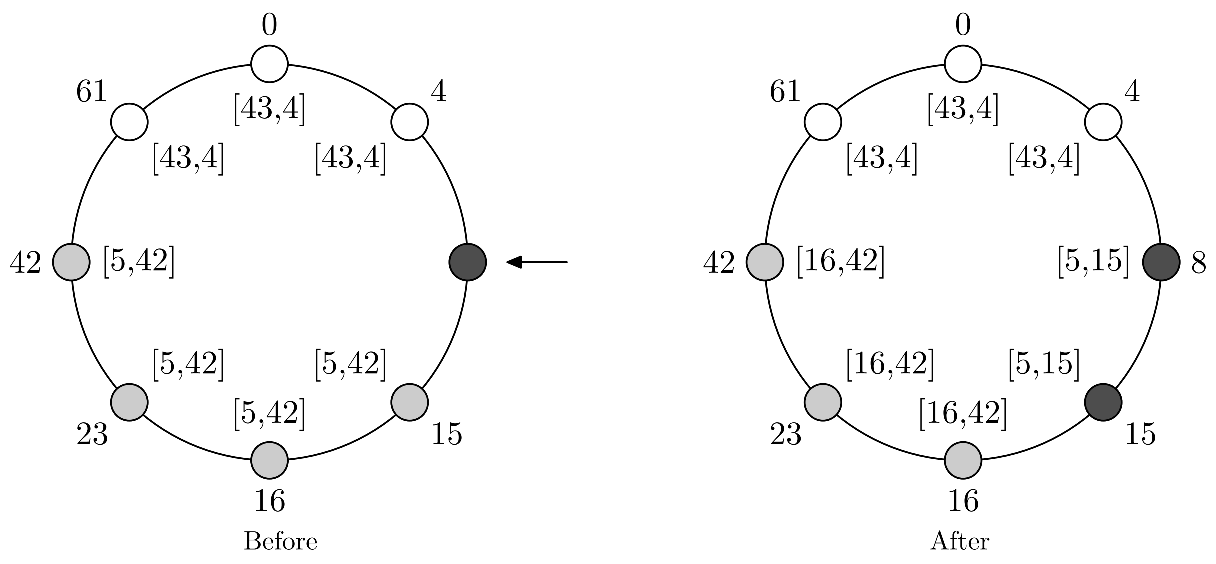


Figure 4.2. Agent join operation

42). Once the message reaches an agent whose key-range includes 28, that agent forwards the message directly to all other agents in its neighborhood. These are all agents in the ring whose key-ranges include 28.

Network Dynamics

To maintain the invariant that the number of score-managers for each key-range stays between k and c, a Penny network must occasionally split or merge neighborhoods as agents join and leave the network. If a peer-join causes a neighborhood's population to rise above c, it splits into two smaller neighborhoods. Dually, if peer-leave reduces a neighborhood's population below k, some or all peers from an adjacent neighborhood migrate in.

When an agent a_{new} joins a Penny ring, it is by default assigned a key-range identical to its successor's. Its successor informs all agents in its neighborhood that they should update their finger tables to include a_{new}. However, if this would result in a neighborhood size b that is greater than c, a split occurs. The first $\lfloor b/2 \rfloor$ agents and the last $b - \lfloor b/2 \rfloor$ agents in the neighborhood each become their own neighborhoods. The key-ranges of the new neighborhoods are the unions of the identifier ranges of the agents within each.

Figure 4.2 illustrates a join operation with a split. Identifiers are labeled next to each agent outside the ring, and agent key-ranges are labeled inside the ring. In this example, $k = 2$, $c = 4$, and $m = 6$, so when the agent with identifier 8 joins, key-range $[5, 42]$ has more than c agents and must be split.

When an agent a_{old} leaves a Penny ring, it informs its successor a_{succ} and the other agents in a_{old}'s neighborhood. If a_{succ} is in a different (adjacent) neighborhood, a_{succ} informs the other agents in that neighborhood that the neighborhood's key-range has grown to include identifiers up to and including $id_{pred} + 1$ (where a_{pred} is a_{old}'s predecessor). Likewise, agents in a_{old}'s neighborhood must shrink their key-ranges so that they end with id_{pred}.

If the departure of a_{old} causes a_{old}'s neighborhood to have fewer than k members, two adjacent neighborhoods must be merged. Let H_{old} be a_{old}'s neighborhood and H_{pred} be the preceding neighborhood. If $|H_{old}| < k$, then the agent in H_{old} whose predecessor is in H_{pred} sends a merge request to its predecessor. That merge request is then forwarded to all agents in H_{pred}. If $|H_{pred}| \leq k + 1$, then both neighborhoods merge to form a single neighborhood. Otherwise, the rightmost $(|H_{pred}| - |H_{old}|)/2$ agents of neighborhood H_{pred} join neighborhood H_{old}. The key-ranges of the new neighborhoods are the unions of the identifier ranges of the agents in the new neighborhoods.

Figure 4.3 illustrates an agent leave operation that requires a key-range merge. Here, the departure of agent 15 from the ring leaves fewer than $k = 2$ agents in its neighborhood. Agent 16 therefore merges with its predecessor neighborhood; agents in both neighborhoods extend their key-ranges to include the identifier ranges of all agents in the new neighborhood.

Whenever an agent's key-range shrinks due to any of the above operations, it must transfer any state associated with keys not in its new range to the appropriate key-holders. Similarly, whenever its key-range grows, it receives state associated with new keys from the agents who previously occupied that range. An average net population change of $\frac{1}{2}(c - k)$ agents per neighborhood is required before that neighborhood will need to be split or merged.

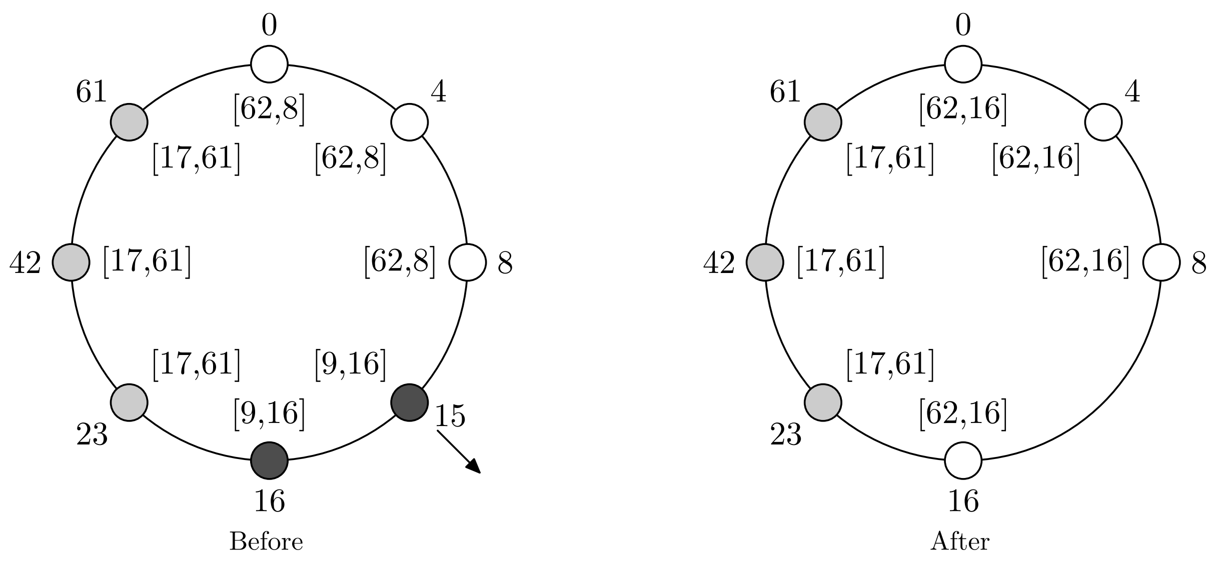


Figure 4.3. Agent leave operation

Thus, by initializing c to be large relative to k, the frequency of these state transfer operations can be reduced.

Agent's Local State

In addition to routing messages, each agent a in a Penny network plays three different roles: It acts as a server when sharing objects, as a score-manager for agents whose keys fall within its key-range, and as a key-holder for objects whose keys fall within its key-range. For each of these roles, it maintains some internal state:

- To act as server, it maintains a list of the identifiers id_o of each object o that it owns.
- To act as score-manager, it maintains a list of daughter agents a_d that satisfy $key_{a_d} \in kr(a)$. These are the agents for whom agent a is a score-manager. For each daughter agent a_d, it also maintains a vector of global trust values T_{a_d} with global integrity and confidentiality labels i_{a_d} and c_{a_d}, respectively.
- To act as key-holder, it maintains a list of daughter objects o_d that satisfy $key_{o_d} \in kr(a)$. These are the objects for which agent a is a key-holder. For each daughter object o_d,

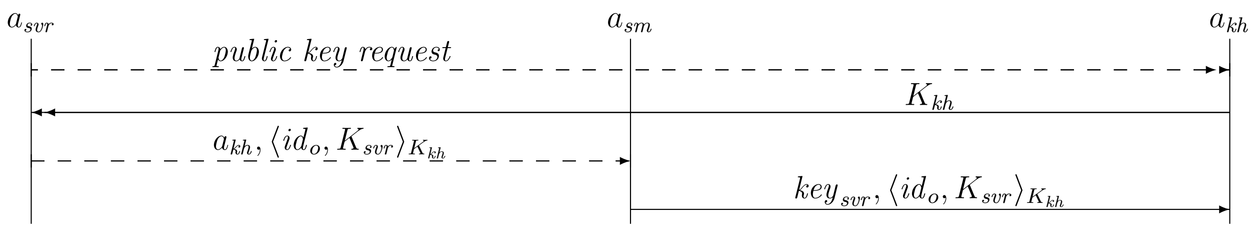

a_{svr}
a_{sm}
a_{kh}
public key request
K_{kh}
$a_{kh}, \langle id_o, K_{svr} \rangle_{K_{kh}}$
$key_{svr}, \langle id_o, K_{svr} \rangle_{K_{kh}}$

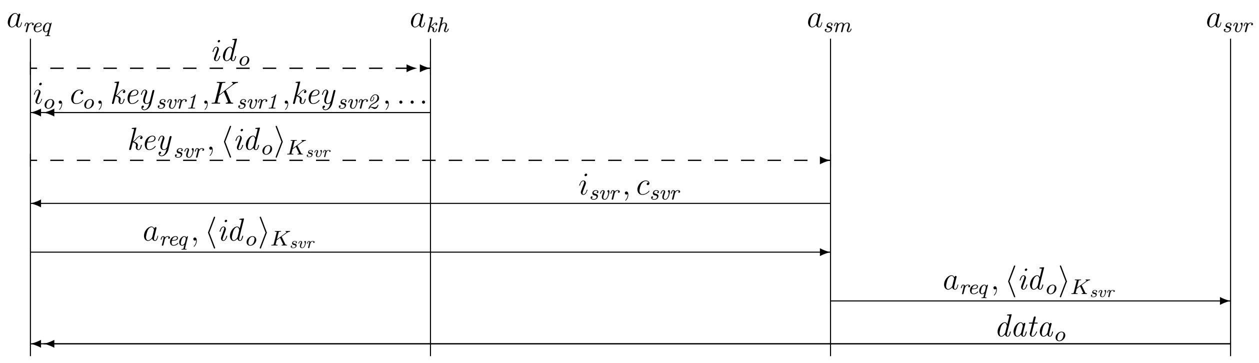
Figure 4.4. Publish protocol for traditional file objects

it maintains a vector of global trust values T_{o_d} with global integrity and confidentiality labels i_{o_d} and c_{o_d}, respectively.

- For encrypted communication, it chooses a public key, private key pair (K_a, k_a).
- It maintains a list of the keys key_{svr} and public keys K_{svr} of the agents that serve object o. Thus key-holders do not learn the actual identifiers of agents who serve object o, only their keys.
- It maintains local trust values $t_a(a_1)$ and $t_a(o)$ for agents a_1 and objects o with whom it had experience. These local trust values give rise to local integrity and confidentiality labels that agent a associates with a_1 and o.

Publishing and Downloading Protocols for Traditional File Objects

Once a Penny network has been initialized, agents interact according to the protocols detailed below. The protocol diagrams that follow use solid arrows to denote messages that are sent directly from agent to agent without using the P2P overlay, and dashed arrows for messages that use the P2P overlay to find the message target based on its ring identifier. Dashed arrows therefore actually involve sending $O(\log N + k)$ total messages. Arrows with double-heads may optionally be sent via anonymizing tunnels for privacy (Chaum and van Heyst, 1991; Freedman et al., 2002; Zhu and Hu, 2008). Notation K_a denotes agent a's public key, and $\langle \ldots \rangle_K$ denotes a message encrypted with key K.



a_{req}
a_{kh}
a_{sm}
a_{svr}
id_o
$i_o, c_o, key_{svr1}, K_{svr1}, key_{svr2}, \ldots$
$key_{svr}, \langle id_o \rangle_{K_{svr}}$
i_{svr}, c_{svr}
$a_{req}, \langle id_o \rangle_{K_{svr}}$
$a_{req}, \langle id_o \rangle_{K_{svr}}$
$data_o$

Figure 4.5. Request protocol for traditional file objects

When an agent a_{svr} wishes to share an object o, it must first publish that object according to the protocol depicted in Figure 4.4. Agent a_{svr} first obtains (possibly anonymously) the public keys of all key-holders a_{kh} for object o. Agent a_{svr} next encrypts the object identifier and its own public key with each of the key-holders' public keys. It asks one of its score-managers a_{sm} to forward the encrypted messages to the key-holders a_{kh}. Agent a_{sm} conceals agent a_{svr}'s identity by sending only its key (which later can be used to get the global trust values and labels from the server's score-manger) to the key-holder rather than its identifier, along with the encrypted message.

To request an object (Figure 4.5), requester a_{req} first sends the requested object's identifier to all key-holders a_{kh} for the object. Each key-holder responds with the object's global integrity and confidentiality labels, and a list of the keys and public keys of servers who offer the object. Agent a_{req} can then obtain the object from any server a_{svr} by sending a request to all score-managers for agent a_{svr}. Score-mangers reply to a_{req} with the server's global trust labels. Based on a selection procedure (Section 4.1.2), a_{req} then sends a download request message. In the message, the requested object's identifier is encrypted with the server's public key to avoid disclosing it to the selected server's score-manager. The score-managers forward the request to the server. The server can then anonymously send the data directly to the requester.

Publishing and Downloading Protocols for RDF Datasets

Besides traditional file lookup, Penny also supports RDF dataset queries. RDF triples are stored within file objects, but it would be prohibitively inefficient and insecure to store all triples within a single file owned by a single peer. Triples are therefore distributed over many smaller files distributed across many peers, with a protocol for locating each triple's containing file. Neighborhoods therefore collaborate to manage a subset of triples. Instead of keys for files, we associate triples with identifiers directly, and all neighborhood agents reply with list of servers who own the identified triples. One is chosen from this list using the selection procedure in Section 4.1.2.

This publishing procedure is detailed in Algorithm 3. To distribute the load of serving particularly popular triples, each agent maintains a usage count for each triple it serves. When this count exceeds an agent-imposed popularity threshold, it defers storage of future instances of that triple component to its successors in the ring. This implements a form of coalesced chaining in the distributed hash table.

RDF queries have syntax `(`[`?`]s`,`[`?`]p`,`[`?`]o`)`, where s is a subject, p is a predicate, and o is an object, and where each optional `?` indicates an unknown in the query. For example, query `(s,p,?o)` requests all RDF triples satisfying subject `s` and predicate `p`. For downloading or querying RDF datasets over Penny network, agents implement Algorithm 4.

Reputation-based Trust Management

Penny incorporates a reputation-based trust management system based on EigenTrust (Kamvar et al., 2003). EigenTrust is a secure, distributed trust management system that maintains a globalized trust value for each agent. These globalized trust values are obtained by an iterative computation that approximates the left eigenvector v of the matrix T of all local trust values in the network. That is, if we define element T_{ij} to be the degree to which agent a_i trusts agent a_j, then the left eigenvector v of matrix T measures each agent a's global trust based on how much each agent trusts a, how much each agent trusts the agents who trust a, etc.

If an agent a_i downloads a file or RDF data from an agent a_j, it rates the transaction as positive (rating 1) or negative (rating -1) based on the experience. We may define local trust value $s(a_i, a_j)$ as the sum of these ratings of agent a_j by agent a_i. Then, in order to aggregate the local trust values, they are normalized. We may define normalized local trust value, $c(a_i, a_j)$, as follows:

$$c(a_i, a_j) = \frac{\max(s(a_i, a_j), 0)}{\sum_x \max(s(a_i, a_x), 0)} \tag{4.1}$$

This ensures that all values are between 0 and 1. These normalized local trust values are then aggregated.

To keep the algorithm scalable and robust, eigenvector v is computed in a distributed and redundant fashion, where k different agents (score-managers) are responsible for computing each element of v. This conforms to Secure EigenTrust (Kamvar et al., 2003), except with global trust labels extended to objects as well as agents, and score-manager replicas grouped into Penny neighborhoods for better performance rather than disbursed throughout the ring.

Data Selection Procedure

Every object request (whether a traditional file download or RDF query) delivers to requesting agent a_{req} a set S of agents who can supply the object. If some respondents are malicious, some of these responses may differ. Agent a_{req} must choose among them based on their reputations. To do so, it partitions S by response. Let R denote the resulting equivalence relation, so that quotient set S/R is the set of agent groups, each of which returned a common response. For each partition $P \in S/R$ we compute the following evaluation function:

$$f(P) = w_1 \frac{|P|}{|S|} + w_2 \frac{\sum_{a \in P} t(a)}{\sum_{a \in S} t(a)} + (1 - w_1 - w_2) \frac{1}{|S/R|} \tag{4.2}$$

Algorithm 3 Publish protocol for RDF data

```
for each RDF triple $r$ do
  for each part (subject/predicate/object) $r_p$ of $r$ do
    $attempt \leftarrow 0$
    while $true$ do
      $id \leftarrow succ(h(r_p + attempt))$
      ask $a_{id}$ to store the triple $r$
      if $a_{id}$ already at popularity threshold then
        $a_{id}$ refuses to store $r$
        $attempt \leftarrow attempt + 1$
      else
        $a_{id}$ stores $r$
        break
      end if
    end while
  end for
end for
```

Algorithm 4 Download protocol for RDF data

```
for each sub-query $q$ in query $Q$ do
  for each part (subject/predicate/object) $q_p$ in $q$ do
    $attempt \leftarrow 0$
    while $true$ do
      $id \leftarrow succ(h(q_p + attempt))$
      Request triples $D$ from $a_{id}$ satisfying $q_p$
      if $|D| <$ agent $a_{id}$'s popularity threshold then
        break                                    // the search for $q_p$ is finished
      else
        $attempt \leftarrow attempt + 1$
      end if
    end while
  end for
end for
Download triples from servers in list obtained above
Locally compute query result from retrieved data
```

where w_1 and w_2 are weights in $[0, 1]$ that prioritize each partition's relative size and reputation, respectively, in the evaluation. We determine acceptable values for w_1 and w_2 experimentally in Section 4.2.

Algorithm 5 Data server selection procedure

```
if all members of $S$ have trust 0 then
  select one server from $S$ randomly
else if transaction is a police transaction then
  $w_1 \leftarrow 0$
  $w_2 \leftarrow 0$
  for each partition $P \in S/R$ do
    choose partition $P$ with probability $f(P)$
  end for
else
  $w_1 \leftarrow 0.2$
  $w_2 \leftarrow 0.8$
  $B \leftarrow \arg\max_{P \in S/R} f(P)$
  $B \leftarrow \arg\max_{P \in B} |P|$
  choose a partition randomly from set $B$
end if
```

Equation 4.2 is used by Algorithm 5 to resolve the selection choice. In the algorithm, *police transactions* are non-user transactions submitted by the security system during idle times in order to improve convergence. These are discussed in the next section.

4.2 Implementation and Results of Experimental Evaluation

We developed a Java implementation of Penny and tested its ability to weather several simulated attack scenarios. Efficiency and robustness of the network was evaluated in terms of the percentage of successful query responses. All experiments employ 20% malicious nodes and 10 pre-trusted agents (Kamvar et al., 2003, §4.5). Throughout the simulation, we use SHA-256 for all hashing and 2^{160} for the identifier space size. We do not simulate the details of the underlying network and encryption operations, or anonymizing tunnels, since these are covered by prior works (see Section 7.3). Experiments are conducted under dynamic conditions, including peer joins and leaves, and neighborhood splitting and merging.

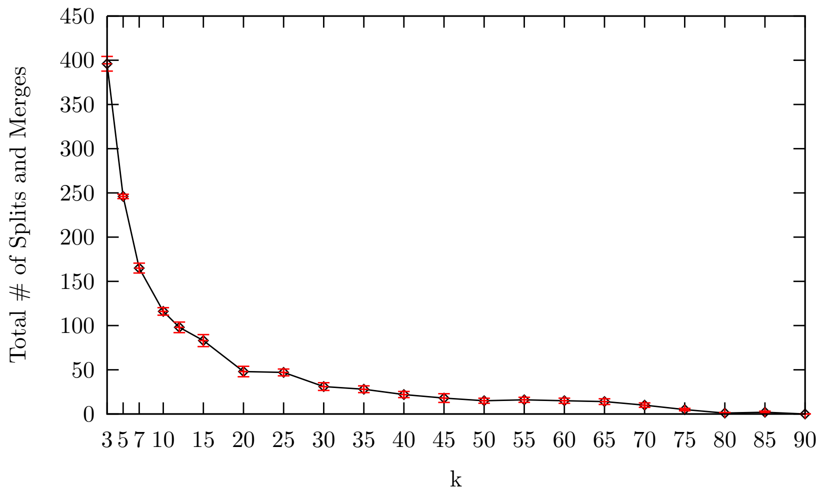
In our implementation and analysis of Penny, we focus on four classes of attacks:

- A malicious agent or collective might spread corrupt or incorrect data. For example, the malicious agent or collective might spread malicious code or circulate false facts.
- A malicious agent or collective might attach incorrect security labels to data. In particular, low-integrity data might be ascribed a high-integrity label, or high-confidentiality data might be ascribed a low confidentiality label.
- A malicious agent or collective might attempt to learn which agents own certain data, perhaps as a prelude to staging additional attacks against those agents.
- A malicious agent or collective might attempt to generate a list of all data served by a particular agent, violating that agent's privacy.

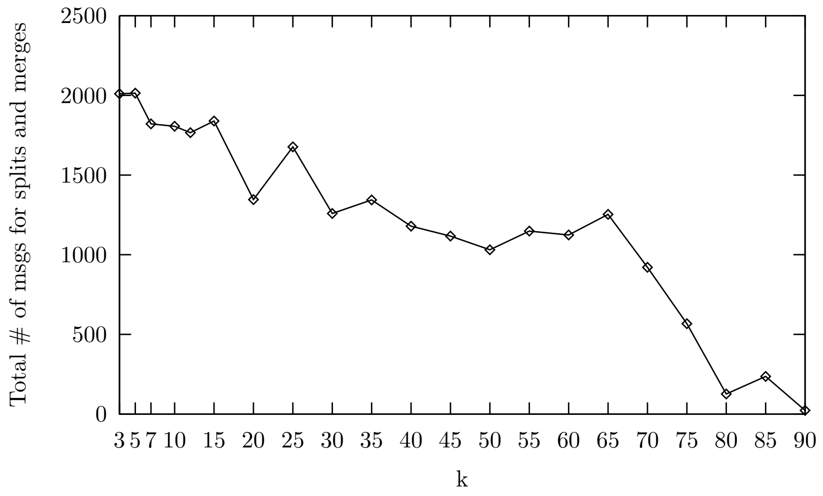
We do not consider attacks upon the network overlay itself, such as message misrouting, message tampering, or denial of service attacks. These attacks are beyond the scope of this work, but could be addressed with various techniques, such as digital signatures, delivery receipts, and non-deterministic routing (Hamlen and Hamlen, 2012).

4.2.1 Bounding Neighborhood Size

As discussed in Section 4.1.2, the upper bound c for the size of a neighborhood must be chosen so as to balance high security (high c) with good performance (low c). We empirically determine a suitable value for c as follows. Each experiment consists of three phases of dynamic activity: (1) 1000 agents join the network, (2) 1000 random joins and leaves occur, and (3) 2000 more joins and leaves occur. All other network activities, including neighborhood splitting and merging, agent finger table updates, periodic EigenTrust runs, file downloads, etc., all occur randomly within all phases. The first two phases serve to initialize and stabilize the network; statistical results are gathered and reported only for phase 3. We tested networks with replication factors k ranging from 3 to 90, with 10 trials per replication



(a) splits and merges ($c = 2k$)

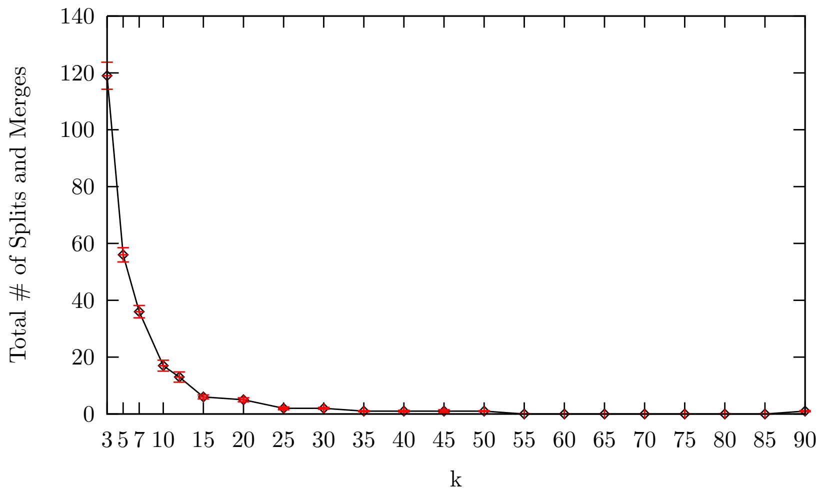


(b) split/merge messages ($c = 2k$)

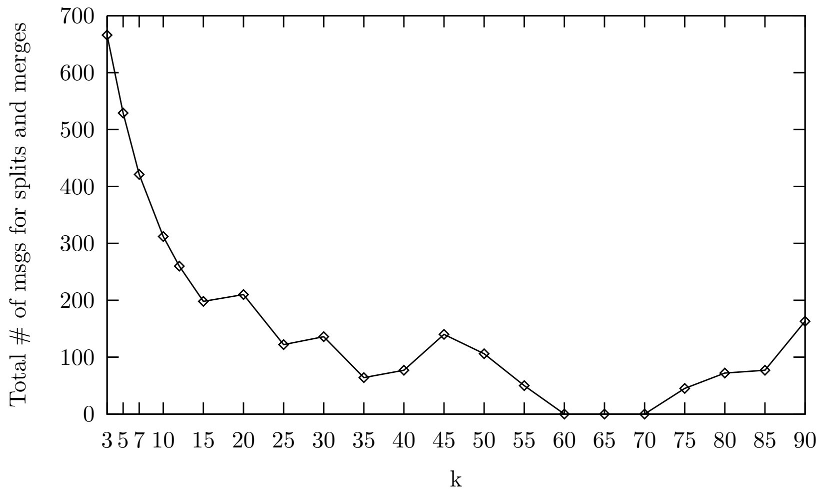
Figure 4.6. Performance comparisons for $c = 2k$

factor. We also tested neighborhood size bounds of $c = 2k$ and $c = 3k$, obtaining the best results for $c = 3k$.

Figures 4.6(a) and 4.7(a) show that the number of neighborhood split and merge operations is greatly reduced when c is increased from $2k$ to $3k$. This is because splitting a size-$2k$ neighborhood results in one of size k, which is near the lower bound on neighborhood size. The new neighborhoods are therefore susceptible to merging, leading to oscillations between sizes k and $2k$, and many expensive merge/split operations. Choosing $c = 3k$ resolves



(a) splits and merges ($c = 3k$)



(b) split/merge messages ($c = 3k$)

Figure 4.7. Performance comparisons for $c = 3k$

this problem. Splitting size-$3k$ neighborhoods yields size-$\frac{3}{2}k$ neighborhoods, which must undergo considerable churn before they must be merged (at size k) or split again (at size $3k$). Even though the neighborhoods are larger, the vastly reduced number of split/merge operations leads to significantly fewer maintenance messages total, as shown in Figures 4.6(b) and 4.7(b). The curve in Figure 4.7(b) is smoother than the one in Figure 4.6(b) because of the elimination of the oscillations.

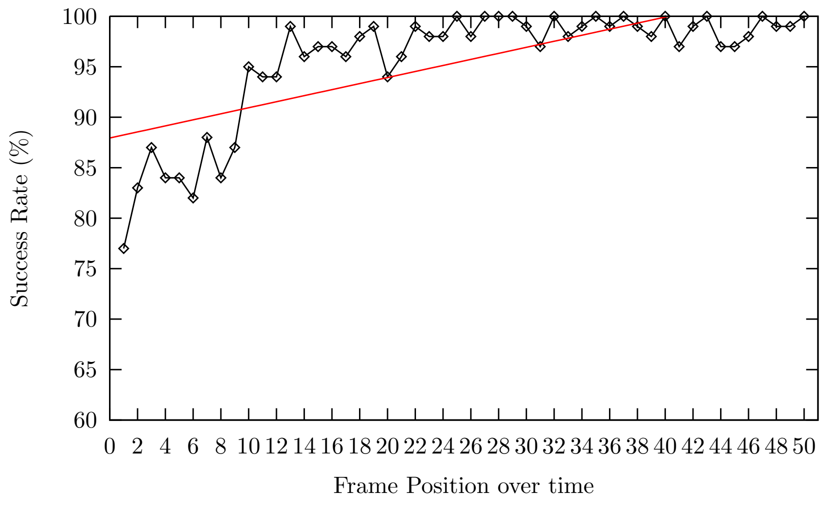
4.2.2 Results for File Downloads

In this section we conduct different experiments for traditional file downloads in the presence of malicious agents, and show the robustness of Penny against these attacks. We simulate the publish protocol (Figure 4.4) and the request protocol (Figure 4.5). For these experiments, there are 1000 agents and 100 file objects in the system, and $k = 5$. We run 1000 downloads in the simulation, each of which uses the selection procedure in Algorithm 5.

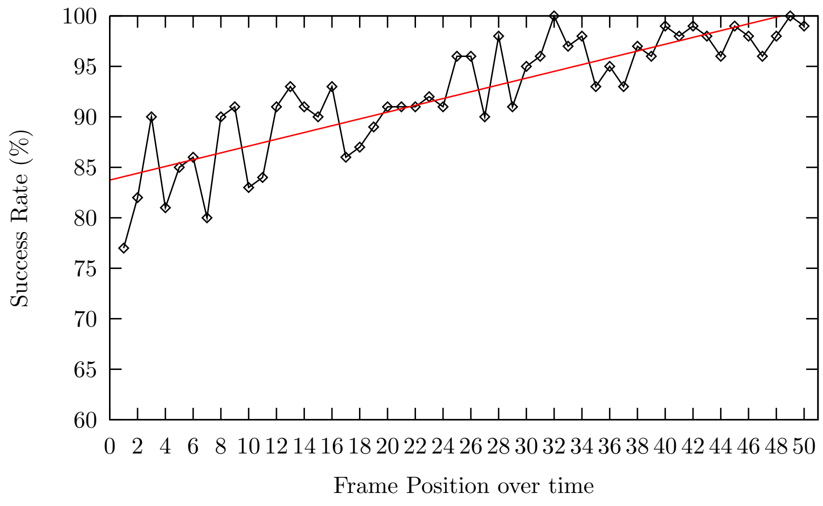
Algorithm 5 makes the natural choice of preferring high- over low-reputation agents for user-submitted requests. We discovered that this tends to cause EigenTrust (and other reputation-based trust management systems) to converge slowly because low-reputation agents are so rarely exercised. To correct this, we introduced a new form of transaction, called a *police transaction*, that is designed to harmlessly exercise the system during idle periods rather than yield a correct result. Such transactions utilize low-reputation agents, providing higher-reputation agents additional opportunities to evaluate their answers. In our simulations, we used 50% police transactions.

For non-police transactions, we placed greatest weight on reputations ($w_2 = 0.8$) and the remaining weight on consensus size ($w_1 = 0.2$). We consider each 20 downloads as one frame and thus show the frame position over time with 1000 downloads. After each frame, we run the EigenTrust algorithm and compute global trust values accordingly. For each type of experiment, we run it 5 times and take the average success rate. For all experiments, we pessimistically assume that all malicious agents know the identities of all the pre-trusted agents, and that they must display high trust for those agents in order to avoid lowering their own reputations. Thus, malicious agents trust only other malicious agents and pre-trusted agents.

In our *negative feedback experiment*, malicious agents always serve malicious files, and non-malicious agents who download the files always submit negative feedback for the transaction. Figure 4.8 shows that under these conditions, malicious agents fail to accrue high



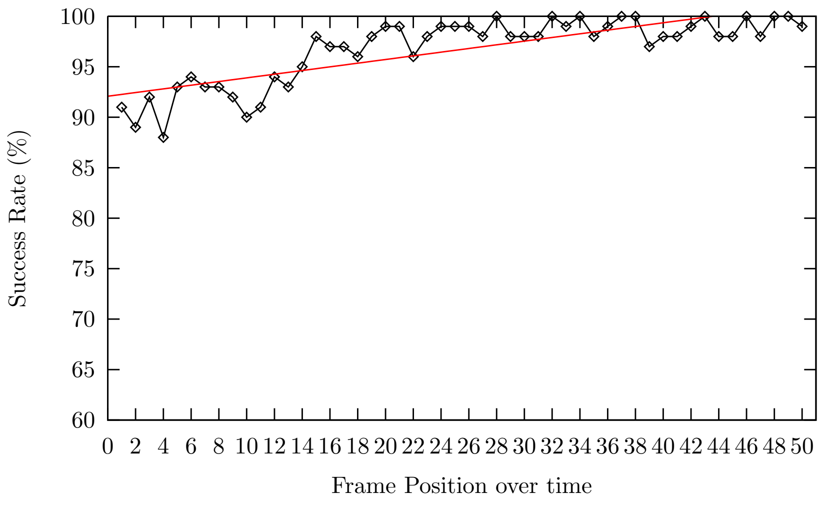
(a) static network



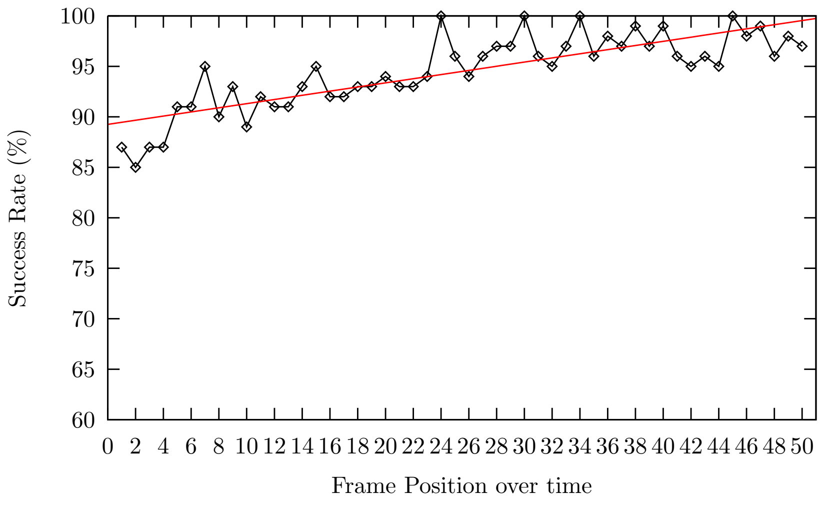
(b) dynamic network

Figure 4.8. Negative feedback experiment success rates

trust. Figure 4.8(a) is for a static network with no leaves or joins, and Figure 4.8(b) is for a dynamic network undergoing constant churn. As expected, convergence is slower in the presence of dynamic activity; the static network converges at about frame 10, whereas the dynamic doesn't until about frame 20. For both, we get a very high average success rate: 95.58% for the static network and 92.22% for the dynamic one, even with 20% malicious agents.



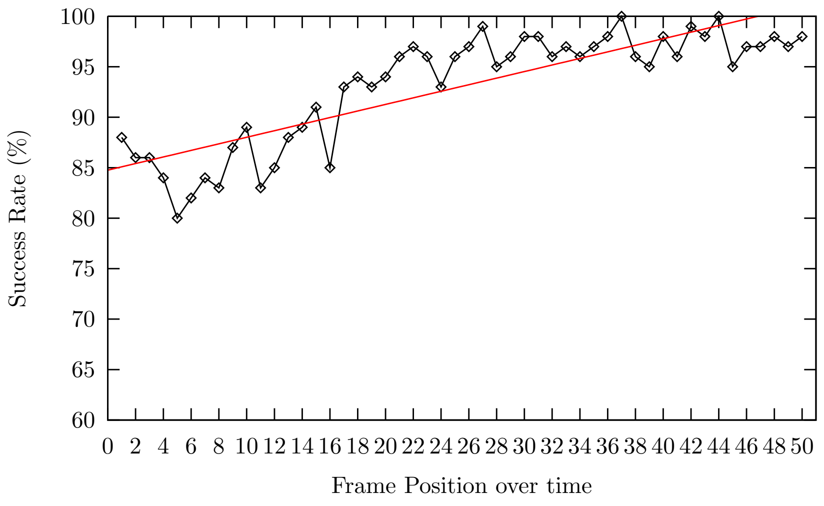
(a) static network



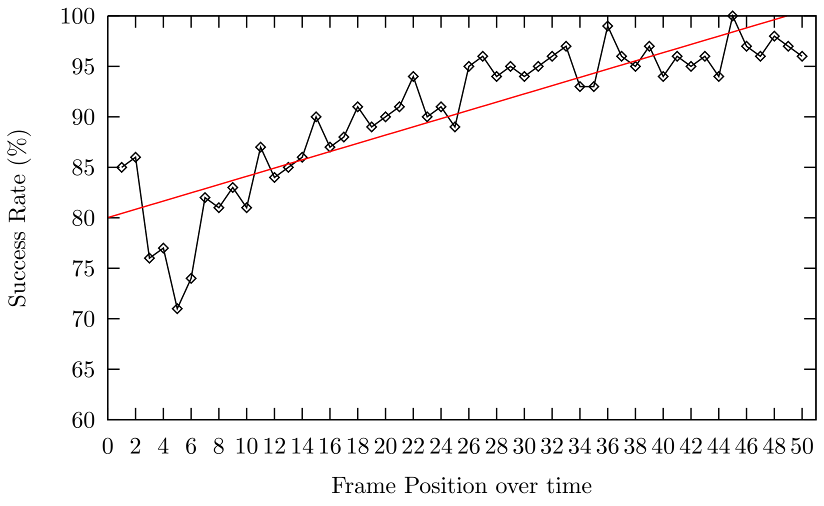
(b) dynamic network

Figure 4.9. Half-correct behavior experiment success rates

Figure 4.9 records the results of our *half-correct behavior experiment*, in which malicious agents provide correct files 50% of the time. Non-malicious agents always provide positive feedback for correct files and negative feedback for corrupt ones. Both static and dynamic networks converge quickly—at approximately frames 14 and 24, respectively. Average success rates were also still very high: 96.72% for the static network and 94.50% for the dynamic one. We further observe that the success rates are higher than each corresponding negative feedback experiment, since malicious agents provide correct files 50% of the time. On the



(a) static network, success rate

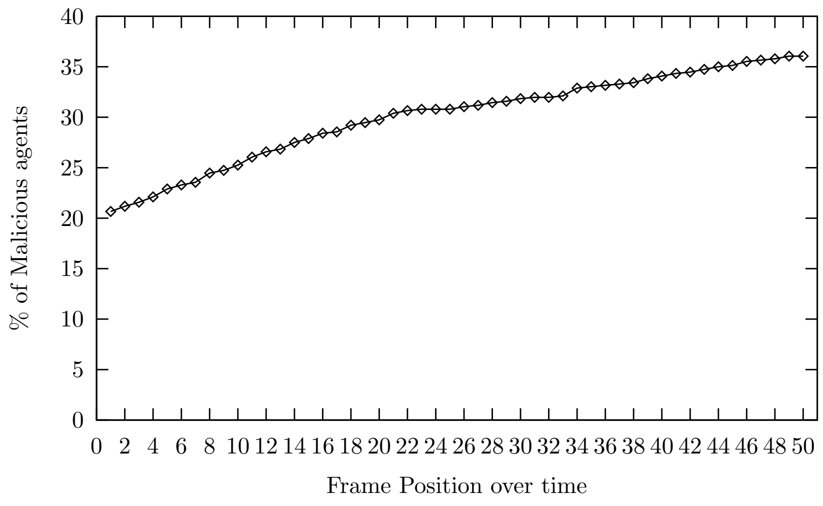


(b) dynamic network, success rate

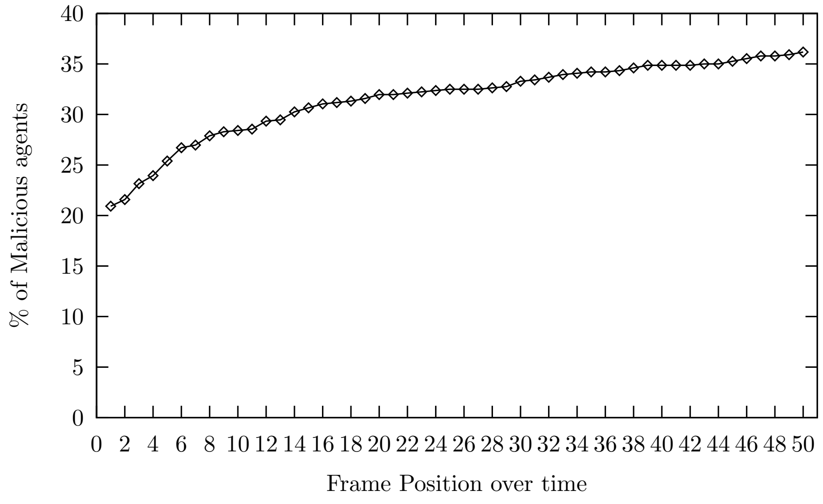
Figure 4.10. Malware propagation experiment success rates

other hand, convergence is slower because non-malicious agents take longer to identify the malicious agents.

Our *malware propagation* experiment next considers the pervasive problem of botnet malware infections of P2P file-sharing networks. In this experiment, non-malicious downloaders of malicious files have a 20% chance of becoming infected and exhibiting malicious behavior thereafter. Malicious agents behave the same as in the half-correct behavior experiment. In both static (Figure 4.10(a)) and dynamic (Figure 4.10(b)) networks, success rates ini-



(a) static network, propagation rate



(b) dynamic network, propagation rate

Figure 4.11. Malware propagation rates

tially drop as previously high-reputation agents suddenly attack the system. However, the reputation system adapts and around frame 16 the non-malicious agents manage to largely isolate the infection. The count of malicious agents continues to grow monotonically, as seen in Figures 4.11(a) and 4.11(b), because the experiment includes no facility for disinfection. But the growth slows, and any new malicious agents are identified relatively quickly by the non-malicious majority. The average success rates were 93.04% for static networks and 90.44% for dynamic ones.

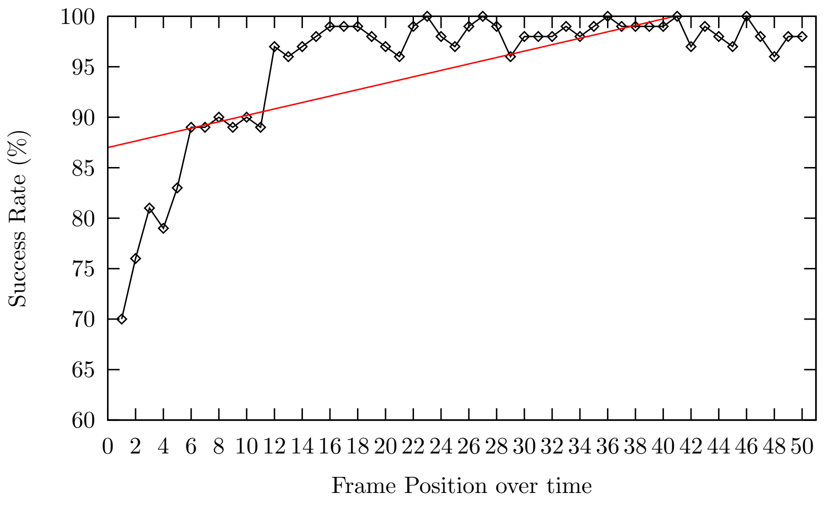
4.2.3 Results for RDF Datasets

We next present experiments for RDF dataset downloads in the presence of malicious agents, and show the robustness of Penny networks. We simulate the publish protocol (Algorithm 3) and download protocol (Algorithm 4). The rest of the experimental setup is same as in Section 4.2.2. We use the LUBM100 (Guo et al., 2005) dataset for our experiments, which is broadly used by researchers for similar evaluations (Guo et al., 2004). The LUBM data generator yields datasets in RDF/XML format, which we converted to N-triples format. For download or query purposes, we use atomic triple queries and conjunctive multi-predicate queries (cf., Cai et al., 2004). We conduct the same three sets of experiments for RDF datasets as reported in Section 4.2.2.

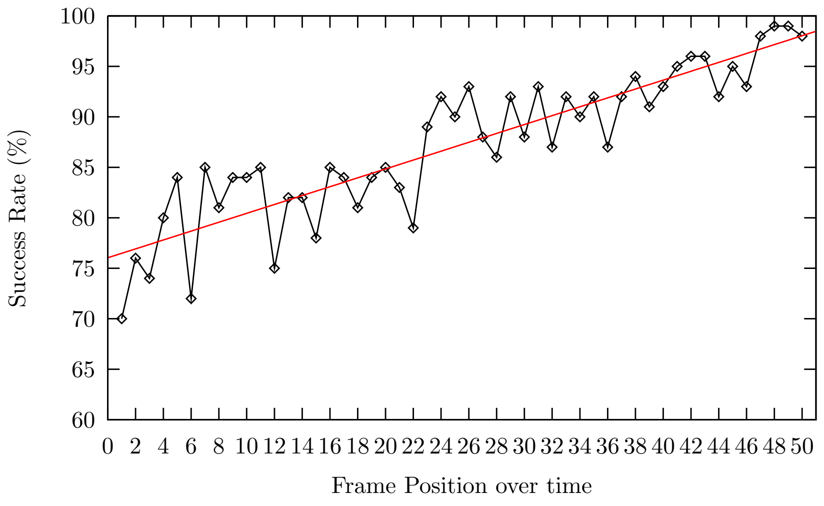
For the negative feedback experiment (Figure 4.12) we see average success rates of 95.12% for static networks and 87.26% for dynamic ones. These are slightly lower than the corresponding rates for non-RDF file downloads because of the additional number of transactions required to successfully answer RDF queries. If any sub-query fails, the entire query fails. In addition, the coalesced chaining implemented by Algorithm 4 requires additional transactions to retrieve popular triples. Convergence rates are slightly lower for the same reason. Despite this, both success rates and convergence rates remain quite high for a network with so much malicious population.

The half-correct behavior experiment exhibits even faster convergence, as seen in Figure 4.13. The static network converges at about frame 15, and the dynamic at 25. Average success rates were similarly high at 96.46% and 92.78%, respectively.

While malware is not possible in RDF data to our knowledge, for the sake of completeness we replicated the malware propagation experiment for the RDF publish and download protocol. Results are reported in Figures 4.14–4.15. Both static and dynamic networks exhibited fast convergence; about frame 19 for the static network and 29 for the dynamic one. Success rates were similarly promising, being 92.90% and 88.98% on average for the



(a) static network



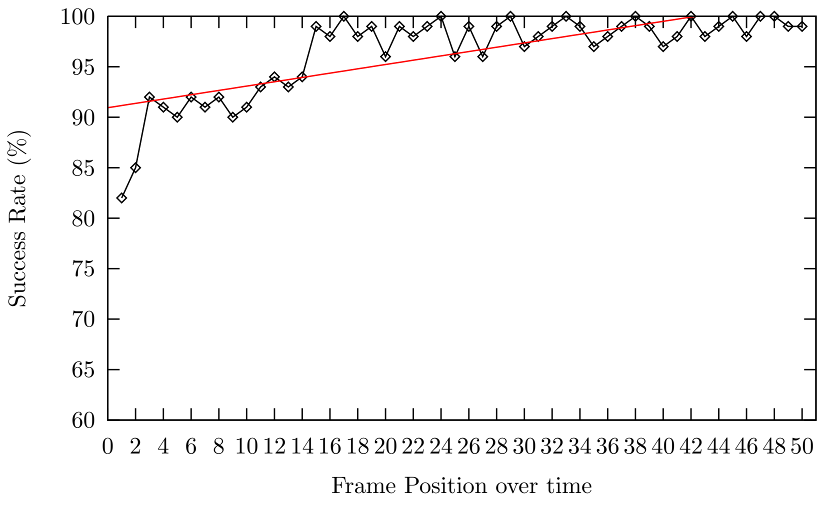
(b) dynamic network

Figure 4.12. RDF negative feedback experiment success rates

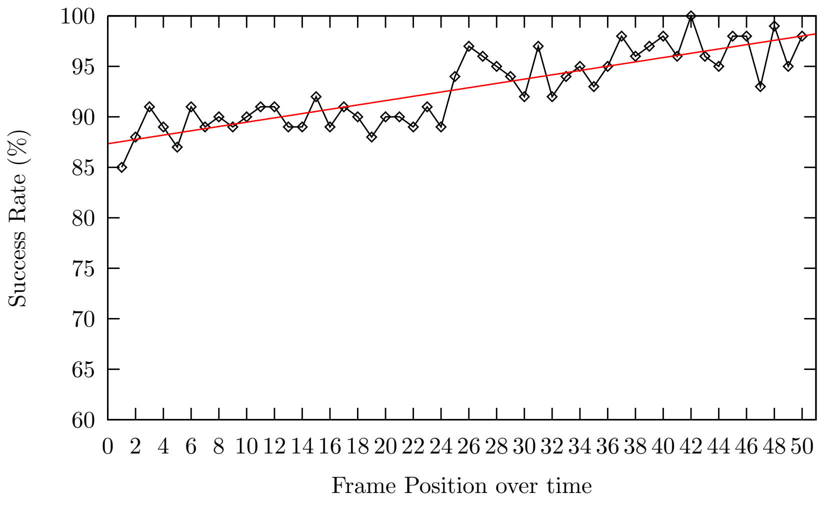
static and dynamic cases, respectively. Again, these are slightly lower than for file downloads because of the higher complexity of the RDF protocol. As before, both networks exhibit an initial drop in success but manage to adapt and recover fairly smoothly.

4.3 Discussion

The high success rates and strong convergence properties experimentally observed in Section 4.2 can be traced largely to Penny's support for exceptionally high data replication via



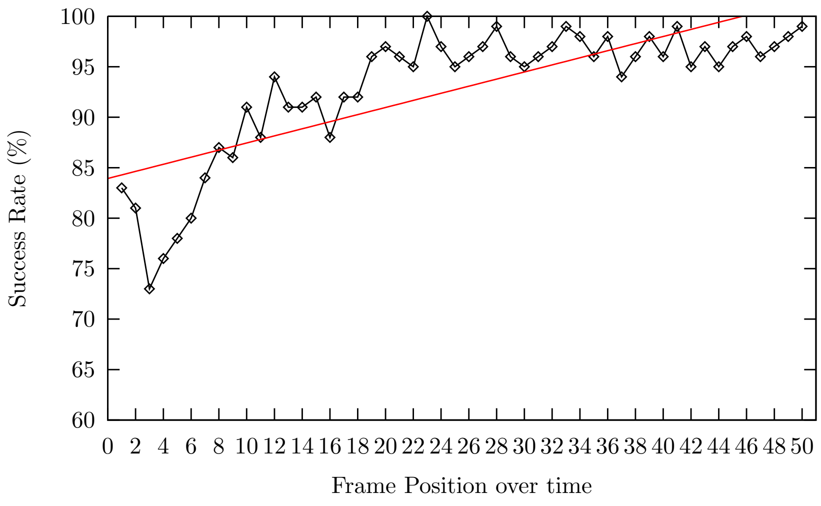
(a) static network



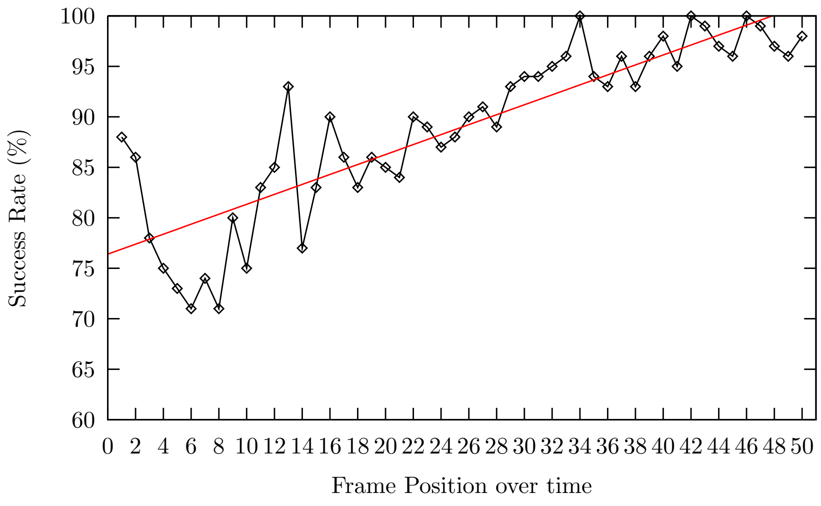
(b) dynamic network

Figure 4.13. RDF half-correct behavior experiment success rates

its neighborhood topology. Label retrieval is efficient in Penny, requiring approximately the same number of messages as object lookup in a Chord network, but with k independent replicas of each label. An agent can retrieve any object's global integrity label by sending a single request message, which gets forwarded at most $O(\log N + k)$ times throughout the network. The request solicits $O(k)$ response messages, from which one response is selected via Algorithm 5.



(a) static network, success rate

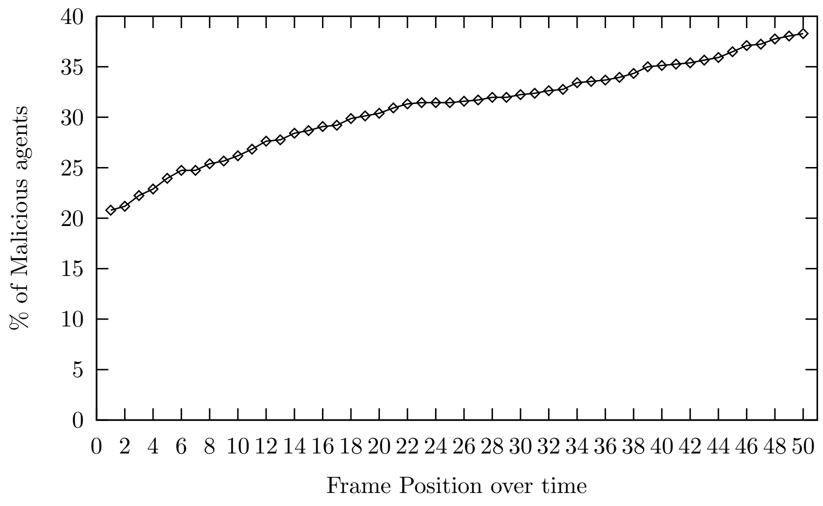


(b) dynamic network, success rate

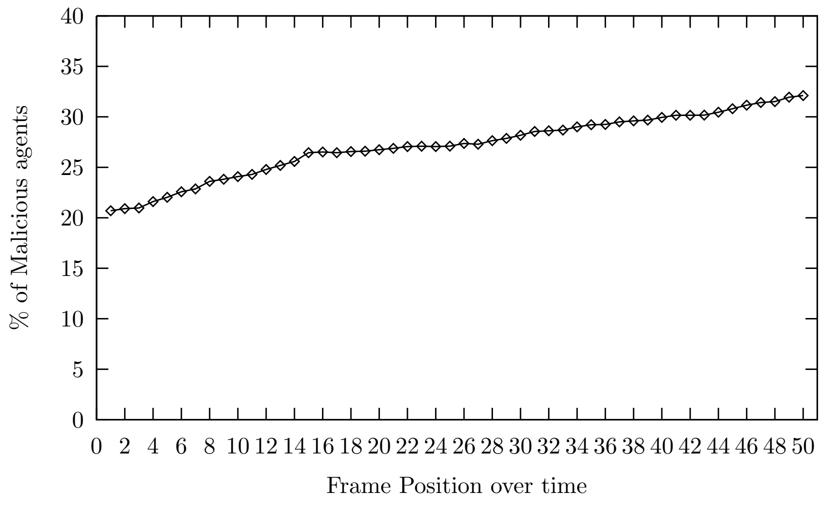
Figure 4.14. RDF malware propagation experiment success rates

Penny inhibits the spread of low-integrity data (e.g., malware) by maintaining a global integrity label for each object shared over the network. Agents wishing to avoid such data can therefore consult each object's global integrity label before downloading it. Thus, the problem of restraining the spread of malware over a Penny network reduces to the problem of efficiently maintaining and reporting accurate integrity labels.

In addition to global integrity labels, Penny also maintains global confidentiality labels for objects. Agents can use these labels as a basis for selectively serving data to other



(a) static network, propagation rate



(b) dynamic network, propagation rate

Figure 4.15. RDF malware propagation rates

agents—possibly based on the requester's trust level, global confidentiality label, or other credentials.

An object's global security labels are determined by the votes of other agents in the network via EigenTrust (Kamvar et al., 2003). Votes are weighted by the reputation of each voter so that the votes of agents who are widely regarded as trustworthy are more influential than the votes of those who are not. This makes it difficult for a malicious agent to attach a high integrity label to low-integrity data. In order for such an attack to succeed, malicious agents must collectively have such good reputations that they outweigh the votes of all other voters. Penny uses EigenTrust to track agent reputations and to prevent malicious agents from accruing good reputations.

Secure hashing and replication are both employed to protect against malicious key-holders and score-managers who might falsify an object's global integrity labels or an agent's global trust value. Use of a secure hash function for identifier assignment ensures that agents cannot dictate the set of objects and agents for which they serve as key-holders and score-managers. By ensuring that there exist at least k key-holders and score-managers for every key-range, Penny prevents any one agent from subverting the reputation of any object or agent. At least $\lfloor b/2 \rfloor$ agents in a neighborhood must be malicious in order to subvert a reputation, where $b \geq k$ is the neighborhood size.

Malicious peers cannot elevate their own reputations by switching IP addresses or creating false network accounts because all agent and object reputations start at zero in Penny (cf., Kamvar et al., 2003). An agent or object acquires a positive reputation only by participating in positive transactions with other agents. Agents with established reputations then report positive feedback for those transactions, elevating the new agent's reputation.

Unlike Penny, Chord (Stoica et al., 2001) requires each key-holder to maintain a list of the agents who own the key-holder's daughter objects. These lists are reported to any agent who requests the object, divulging the identities of all agents who own a particular object. To address this privacy vulnerability, Penny conceals information associating agents with the objects they own by splitting that information amongst key-holders and score-managers (see Figure 4.5). A malicious key-holder and a malicious score-manager must therefore collaborate to learn that a particular server owns a particular object. Opportunities for such collaboration are limited because key-holders and score-managers cannot choose their key-ranges. It is therefore unlikely that a malicious collective will occupy both a key-range that includes a particular victim object's key and a key-range that includes a particular victim agent's key (assuming the collective is small relative to the size of the network). Thus, Penny enforces a notion of object ownership privacy.

Key-holders and score-managers can, of course, learn ownership information through guessing attacks, but this is prohibitively expensive when the space of object and agent identifiers is large. For example, a malicious agent a_m can discover whether a particular object o is served by any agent for which a_m serves as score-manager by requesting id_o and comparing the key-holders' responses against its list of daughter agents. However, a_m cannot easily produce a list of all objects served by any of its daughter agents because to do so it would have to search the entire space of object identifiers. Likewise, a_m can discover whether a particular server a_{svr} owns any object for which a_m serves as key-holder. To do so, a_m computes key_{svr} and searches for that key in its list of keys of servers that own a_m's daughter objects. However, a_m cannot easily produce a list of all servers that own any given object because it would have to search the entire space of server identifiers. So a large identifier space provides natural resistance to guessing attacks.

CHAPTER 5

COMPUTATION CERTIFICATION AS A SERVICE IN THE CLOUD[1]

Protecting remote software from corruption by untrusted or malicious host environments has long been an important challenge for Trustworthy Computing (TwC) paradigms, such as mobile devices that mix trusted and untrusted hardware (Vasudevan et al., 2012), and trustworthy grids that distribute computations to remote, untrusted hosts (Cooper and Martin, 2006). In these contexts, *untrusted environments* are computing platforms (e.g., hardware, OSes, and VMs) that have unfettered access to the distributed computations they receive, including the ability to tamper with the mobile code, its program state, and its results. To achieve high reliability and integrity for computations, secure grids must prevent or detect all such tampering for each computation they distribute.

Many existing platforms therefore aggressively apply *remote attestation* technologies to detect and preclude software tampering in untrusted environments (Trusted Computing Group, 2011; Nauman et al., 2010). For example, hardware- and software-based attestation mechanisms evidence the integrity of remote client states through cryptographically signed memory snapshots taken statically or at runtime (Kil et al., 2009; Seshadri et al., 2005). However, code integrity alone does not guarantee that a computation result is correct. For instance, an attacker may run the software without any alterations but still return corrupted results to the requester. Code integrity checking must therefore be coupled with result integrity checking, which usually involves embedding a secret sub-computation that is difficult to reverse engineer and that can be checked by the requester (Falcarin et al., 2005),

[1]© 2013 IEEE/ACM. Reprinted, with permission, from Safwan M. Khan and Kevin W. Hamlen. Computation Certification as a Service in the Cloud. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 434–441, May 2013.

or by refactoring distributed computations into function compositions that can be validated cryptographically (M'Barka et al., 2009; Sander and Tschudin, 1998, 1997).

Unfortunately, all of these approaches require a significant redesign of most software. For example, typical Android apps are not easily modified to contain inextricable, secret computations or cryptographically verifiable compositions. As a result, few mainstream mobile computing devices have adopted these technologies. Moreover, many of these solutions rely on software obfuscation (Ceccato et al., 2013; Collberg and Nagra, 2010), which does not provide rigorous guarantees, since clever attackers can potentially reverse the obfuscation.

In contrast, clouds (Amazon, 2013; Microsoft, 2013; Apache, 2013) are an increasingly popular grid computing paradigm (Buyya et al., 2009) utilized by myriad mobile device architectures (Gerla and Huang, 2012). Clouds offer massive parallelism and potentially high integrity assurance through replication. For example, trust management has been used in clouds to ensure that even if some cloud nodes are malicious, computation results are nevertheless correct with high probability (Khan and Hamlen, 2012b). The favorable business model for cloud-powered mobile apps has led to a rapidly growing mobile cloud market that is expected to exceed $9 billion by 2014 (Holden, 2010).

While clouds offer an attractive means for mobile devices to remote their high assurance computations, cloud-assisted devices still face at least two significant limitations in practice: First, most mobile devices do not have perpetual, continuous access to the cloud. Thus, many of their computations must be carried out using purely local resources, pending cloud access. We observe that such devices would benefit from a *trust-but-verify* model in which computations are initially carried out locally using a potentially untrusted environment (e.g., insecure CPU and storage), trusted for a limited time, but then verified once cloud access is available. For example, a password that unlocks a software product could be verified locally, allowing the software it protects to be used for a limited time, after which the tamper-proof portion of the hardware seeks validation of the password verification computation by the cloud.

Second, although clouds offer high parallelism, most everyday app computations are not highly parallelized and therefore derive little or no benefit from such parallelism. Thus, we seek a computation verification strategy that allows mostly serial computations performed in an untrusted environment to be rapidly validated using the massive parallelism offered by the cloud. Such validation empowers clouds with a new form of Security as a Service (SECaaS) that provides high assurance for local, untrusted, mostly-serial computations.

Our answer to this challenge is a Cloud-based COmputation VERifier (CloudCover) that allows untrusted Java computations to yield a *proof of computation integrity* as a side-effect of the computation. The proof can then be validated against the original code and the computation's result to formally verify that the result is correct. Neither the computation nor the proof (nor their origins) are trusted by CloudCover. A (possibly forged) proof either proves that a given computation results from a given code, or it does not. If the former, the result is correct regardless of where the proof came from; if the latter, the computation, the proof, or both are untrustworthy. Thus, CloudCover can be formalized as *proof-carrying computation*, in the spirit of proof-carrying code (Necula and Lee, 1998).

CloudCover proofs have the advantageous quality that the task of verifying them can be parallelized almost arbitrarily even when the original computation is not parallelizable. Thus, they derive maximal benefit from massively parallel architectures, like clouds. To demonstrate, we implement CloudCover for Hadoop MapReduce (Dean and Ghemawat, 2008), and use it validate non-parallelizable Java computations for message digest generation using SHA-1 (National Institute of Standards and Technology, 1995) and MD5 (Rivest, 1992) cryptographic hash functions. Experimental results indicate that CloudCover scales extremely well, with the only practical limit to parallelization stemming from the fixed overhead of dispatching new mappers and reducers.

Our work therefore offers the first computation integrity validation mechanism that

- requires minimal changes to existing software;

- fully leverages the massive parallelism available from commodity data processing clouds, such as MapReduce;
- provides a tunable range of integrity assurances, from rigorous, absolute assurance to probabilistic assurance (with verification overhead scaling linearly with assurance level); and
- is applicable to everyday mobile app computations, such as those that contain mostly serial computations implemented in interpreted bytecode languages like Java.

Following the constitution of a trustworthy cloud (Chapter 2 , 3 and 4), we can take advantage of it for computation certification as a service from this cloud. We propose this in CloudCover (Khan and Hamlen, 2013a), which decentralizes trust as well; nevertheless, this time on the user side. We have already distributed trust inside the clouds making it secure; now we decentralize the trust outside of the clouds and later distribute the computation verification to this cloud.

5.1 System Overview

5.1.1 Architecture and Threat Model

We consider two possible system architectures, one in which trusted and untrusted devices are physically separate, and one in which trusted and untrusted hardware is co-located on a single mobile device. Both architectures are illustrated in Figure 5.1. In both cases, a resource-impoverished, *trusted component* wishes to distribute a computation to an *untrusted component* that is comparatively resource-rich.

The trusted component cannot efficiently distribute the computation across the cloud directly because (a) it currently lacks cloud access, (b) the cloud's computing power is based on parallelism, which goes unutilized when the computation is mostly serial, and/or (c) the

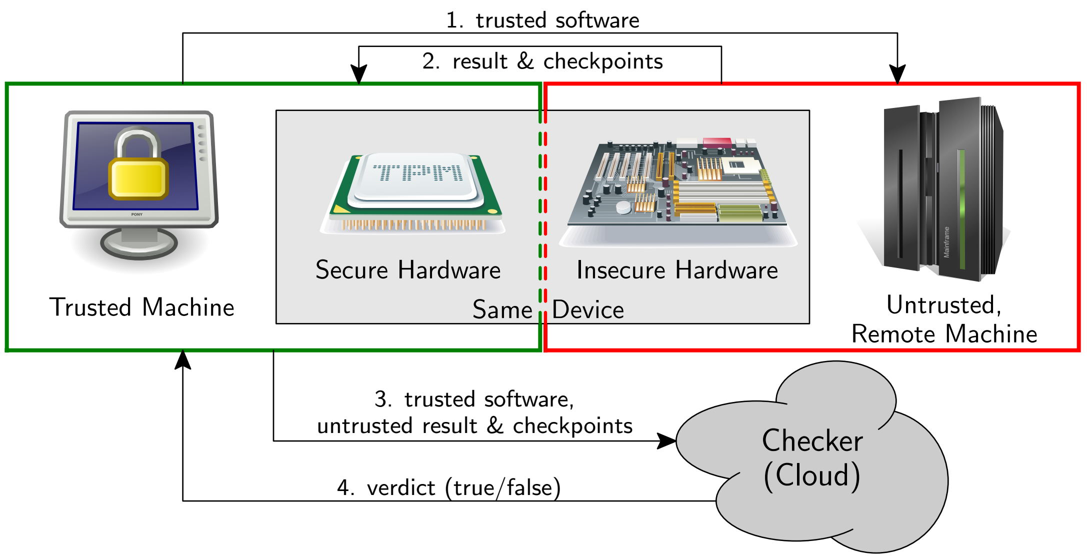


Figure 5.1. System architecture of CloudCover

computation is tentative in the sense that only certain outcomes demand verification, and the computation outcome is not known in advance. The last case arises frequently in grid computing. For example, SETI@home (Anderson et al., 2002) aggressively validates only those computations whose results suggest the existence of extraterrestrial intelligence.

In each scenario, we assume that the trusted component is secure in the sense that no malicious user has access to it, or those that do cannot corrupt its computations, storage, or state. Typically we expect that such components consist of expensive, more secure hardware that is less efficient or more constrained due to its extra security. In contrast, untrusted components consist of insecure hardware and/or remote, untrusted machines that are entirely exposed to malicious users and activities.

The *checker* of our system is a cloud computing platform that may consist of hundreds or thousands of nodes. It is a high assurance, massively parallelized data processing framework whose computation results are trusted, but it is best applied to highly parallelized computations. Assigning it serial computations is prohibitively slow and expensive.

Figure 5.1 summarizes the system workflow together with the architecture. The trusted component first instruments the mobile code with proof-generation logic. The instrumented

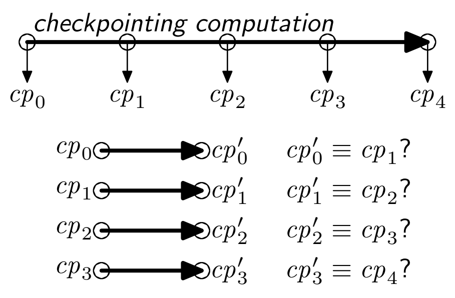

checkpointing computation
cp_0
cp_1
cp_2
cp_3
cp_4
cp_0 cp'_0 $cp'_0 \equiv cp_1$?
cp_1 cp'_1 $cp'_1 \equiv cp_2$?
cp_2 cp'_2 $cp'_2 \equiv cp_3$?
cp_3 cp'_3 $cp'_3 \equiv cp_4$?

Figure 5.2. CloudCover checkpointing and validation

code is then distributed to the untrusted component. A cooperative recipient executes the computation faithfully, yielding a computation result and a proof of computation integrity. The code, result, and proof can then be sent to the cloud (when it becomes available) for parallelized verification. A malicious, untrustworthy, or compromised recipient, however, returns an incorrect result. There exists no proof that the original code yields such a result, so cloud validation inevitably fails, irrespective of the proof submitted by the untrusted component. Thus, the incorrect result is rejected.

5.1.2 Computation Integrity Proof Generation and Validation

CloudCover approaches the problem of proof generation through *checkpointing*, as illustrated in Figure 5.2. Mobile Java code is instrumented with a checkpoint operation that periodically saves the current program state to disk. Initial checkpoint cp_0 is the program start state and is fully characterized by the code itself, so needn't be generated explicitly. The last checkpoint cp_n characterizes the computation result.

A *chain* of such checkpoints constitutes a proof that a computation whose initial state is cp_0 yields result cp_n. The proof can be validated by recomputing all the segments of the chain in parallel. That is, for each checkpoint $i < n$, cloud node i initializes its JVM to state cp_i and computes until the next checkpoint, yielding state cp'_i. It then decides whether $cp'_i \equiv cp_{i+1}$. If any of these equivalence checks fail, the proof fails and the computation is rejected. This transforms a serial computation into a fully parallelized re-computation that only takes as long as the longest checkpoint interval (plus some time for the checkpoint equivalence check) to validate. The equivalence check can be additionally parallelized, as discussed in Section 5.2.

Proof validation through checkpoint chaining engenders a natural trade-off between assurance and computational expense through *spot-checking*. A spot-checking validator recomputes and checks each segment in the checkpoint chain with probability p. This reduces the total computation cost to a fraction p of the total, and detects erroneous computation results with probability p. Thus, clients may tune parameter p in accordance with their desired level of assurance and the expense of cloud computing time.

One naïve way to implement checkpointing for Java programs is to take a system-level snapshot of the JVM process image at fixed time intervals. However, this approach is inadequate for computation certification for at least two reasons: (1) JVM process images vary greatly at the byte level depending on the particular JVM version and the underlying hardware. For example, different JVMs have radically different underlying implementations, including memory allocation strategies, JIT compilation behavior, and a variety of other low-level details. These differences are transparent to Java programs, but they make it difficult or impossible to compare system-level JVM process images for semantic equality. (2) Even when comparing process images from identical JVM versions on identical hardware, semantic equivalence of the JVM state is not an identity function. Many JVM objects have internal fields, such as hash values and clock times, that are irrelevant to semantic object equivalence.

We therefore implement checkpointing at the Java level rather than the system level. Our implementation extends Apache's open source Javaflow library (Commons, 2013). Javaflow includes a *suspend* operation that generates *continuations*. Each continuation object contains a snapshot of the stack trace, including the call stack, local and global variables, and the program counter. Such continuations can later be resumed (i.e., replayed) by passing the continuation object to the library's *continueWith* method. As a simple example, a program that prints numbers from 1 to 100 can be suspended immediately after printing 50. The resulting continuation can be resumed later, resulting in the program printing 51, etc., until another suspension is encountered. Continuations can be serialized for mobile execution. We leverage continuations as checkpoints for CloudCover.

Although Javaflow supports suspension and resumption of computations via continuations, it does not support continuation equivalence-checking. This is necessary for the checkpoint equivalence check in Figure 5.2. CloudCover therefore extends Javaflow's *Continuation* class with an *equals* method that compares two suspended program states for semantic equivalence. Two states are equivalent if they consist of equal-length stacks whose corresponding slots contain equivalent values and objects. Deciding such semantic equivalence is non-trivial in general; for example, the states may contain objects with private fields to which the continuation object lacks access, or they may include fields whose values are semantically equivalent but non-identical. Fortunately, all Java objects have their own *equals* methods, which encode an object-specific notion of semantic equivalence.

Every Java program therefore carries within itself a *general contract* of object-equality-(Bloch, 2008), encoded by the collective implementations of all its *equals* methods. This contract can be leveraged to decide semantic equivalence of arbitrary program states. It is this insight that allows CloudCover to validate computations without any significant change to existing Java programs.

5.1.3 CloudCover Protocol

CloudCover's protocol is detailed in Algorithm 6. We denote the trusted component, untrusted component, and checker as *TC*, *UC*, and *C* (respectively) in the algorithm. The order of the generated checkpoints is important since the checker accepts a checkpoint as input, resumes it, and matches its result with the immediate next checkpoint. If there are n

Algorithm 6 CloudCover Protocol

```
1: $TC$ chooses number $n \geq 2$ and placement of checkpoints
2: $TC$ inserts $n-2$ *suspend* calls into software (the initial and final checkpoints are implicit)
3: $TC$ sends modified (trusted) code $c$ to $UC$
4: $UC$ executes $c$ and sends generated checkpoints and result to $TC$
5: $TC$ sends $c$, checkpoints, and result to $C$
6: $C$ organizes checkpoints into pairs $(cp_i, cp_{i+1})$
7: $C$ dispenses $(c, cp_i, cp_{i+1})$ as input to each *computation unit* (e.g. *mappers* in Hadoop)
8: for each checkpoint $cp_i$ where $i \in [0, n)$ do
9:   if $rand() \leq p$ then
10:    $cp'_i \leftarrow continueWith(cp_i)$
11:    if $cp'_i \equiv cp_{i+1}$ then
12:      computation unit returns *true*
13:    else
14:      computation unit returns *false*
15:    end if
16:  end if
17: end for
18: return the conjunction of all the unit return values
```

checkpoints, it therefore organizes $n-1$ pairs. For line 11, we use our customized *equals()* method included in the modified Javaflow library.

5.1.4 Attacks and Defenses

The following are some ways in which a malicious untrusted component might attack CloudCover:

- Untrusted components can alter the number and/or positions of checkpoints. This is discovered by the checker with probability p when it compares the first altered checkpoint with the one yielded by the trusted software, since their memory states and/or program counters must differ.

- Untrusted components can modify other parts of the software. This threat is certainly detectable since the checker runs the trusted software received from the trusted component, not from the untrusted one, and thus discovers the difference.

- Untrusted components can leave the code uncorrupted, but tamper with the checkpoints and/or results it sends to the trusted component. This is discovered with probability p when a checkpoint equivalence check fails.

CloudCover trusts the cloud platform. Clouds can attain suitable trustworthiness through trust management, replication, virtualization, and a variety of other technologies (e.g., Hatman (Khan and Hamlen, 2012b), AdapTest (Du et al., 2011) or RunTest (Du et al., 2010)) not typically available to mobile devices and other, stand-alone, cloud-assisted machines.

Privacy preservation of computation results is beyond our scope. For such protection, we refer the reader to numerous related works on that subject, including AnonymousCloud (Khan and Hamlen, 2012a), secure multiparty computation (Lindell and Pinkas, 2009), and differential privacy (Roy et al., 2010).

5.2 Implementation

Implementation of CloudCover for a real-world architecture is a key contribution of our work. We therefore target Java computations, which are the basis for many mobile app domains, and we implement computation validation using a commodity data processing cloud—Hadoop MapReduce. We leverage Javaflow's built-in functionalities for taking checkpoints of software and resuming software from checkpoints. Our custom implementation of *equals()* for Javaflow continuations decides semantic equivalence of checkpoints.

The checker is deployed on a Hadoop (Apache, 2013) cluster consisting of 6 DataNodes and 1 NameNode. Node hardware is comprised of Intel Pentium IV 2.40, 3.00GHz processors with 2–4GB of memory each, running Ubuntu operating systems. Javaflow was installed and configured on each DataNode in the Hadoop distributed environment, making it available to distributed jobs. We implemented a mechanism for reading and writing checkpoints for mappers in Hadoop in an appropriate file format for equality-checking with Javaflow. LZO compression was applied to all Hadoop file transfers to minimize transfer and storage costs. For trusted and untrusted components of CloudCover, we use standard desktop computers with configurations similar to the individual cloud nodes above.

For experiments, we select two non-parallelizable cryptographic hash functions, SHA-1 (National Institute of Standards and Technology, 1995) and MD5 (Rivest, 1992), which yield message digests. These were instrumented with Javaflow checkpointing operations placed within their inner loops. Both algorithms are widely used in TwC, yet not parallelizable beyond fine-grained, instruction-level optimization (cf., Nakajima and Matsui, 2002). This makes them good subjects for our tests. Both functions take strings of arbitrary length as input, where SHA-1 and MD5 produce 160-bit and 128-bit message digests, respectively. We choose fairly long strings as inputs in our experiments to demonstrate the benefits of fast, parallelized validation of comparatively long, serial computations.

Aside from verifying checkpoint chain segments in parallel, we additionally parallelized the checkpoint equality checking procedure in our implementation. Continuations are stacks that can be partitioned arbitrarily into sub-stacks that can all be checked in parallel for equivalence. We implemented this for Javaflow by introducing a continuation *compare* method. During comparison, instead of equality-checking each pair of objects inside the checkpoints, a mapper can redirect them to other mappers by submitting new jobs in Hadoop. The advantage is that if any individual checkpoint-pair is extremely large (e.g., very large stacks), then the checkpoint equality-checking job can be parallelized to compensate. In our experiments, the stacks are not that large, so this feature went unexercised. (With small stacks, parallelizing the equality-checking task is not worthwhile, since the task of splitting the stacks introduces more overhead than it saves.)

5.3 Experimental Results

In all our experiments, any corruption of checkpoints (e.g., moving, modifying, or omitting them) and/or corruption of results provoked rejection by the checker (see Section 5.1.4). The remainder of our evaluation therefore focuses on performance.

Our first experiment (Figure 5.3) illustrates the superiority of CloudCover over a single machine verifier for SHA-1 applied to a 38KB input string. We produce 391 checkpoints for this experiment and observe that the Hadoop cluster clearly outperforms a single machine—with 6 nodes, it exhibits approximately 23% gain. Although adding nodes reduces the overall validation time, it suffers diminishing returns typical of parallel architectures. The diminishing returns are primarily due to additional overhead for file I/O on HDFS (Hadoop's file system) and additional network communication per additional node. Additionally, 391 checkpoints for this experiment spawns a number of parallel mappers that exceeds the total capacity of our Hadoop cluster. It therefore places many mappers in the queue. For this reason, a 3-node cluster takes less than double the time of a 6-node cluster (but still beats the single node performance by 10%). With a larger cluster of hundreds or thousands of machines, we therefore expect even better performance than exhibited by our comparatively small-scale testbed.

Our second experiment (Figure 5.4) considers an MD5 algorithm applied to a 38KB input string using the same cluster sizes. It generates 612 checkpoints and achieves a similar performance gain: 3-node and 6-node Hadoop clusters run 12% and 25% faster, respectively, than a single machine setup. The increased number of checkpoints results in longer validation times than the first experiment. This indicates that for any given computation and cluster configuration, there may be an optimal number and distribution of checkpoints. In addition, although the number of checkpoints is 57% greater, the verification time is more than 57% longer than for SHA-1. This is because the MD5 computation's checkpoints are much larger on average than those generated for SHA-1, because the MD5 implementation places more

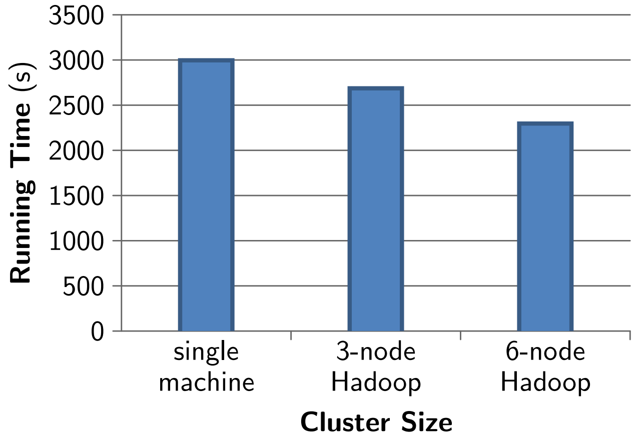


Figure 5.3. SHA-1 computation verification time

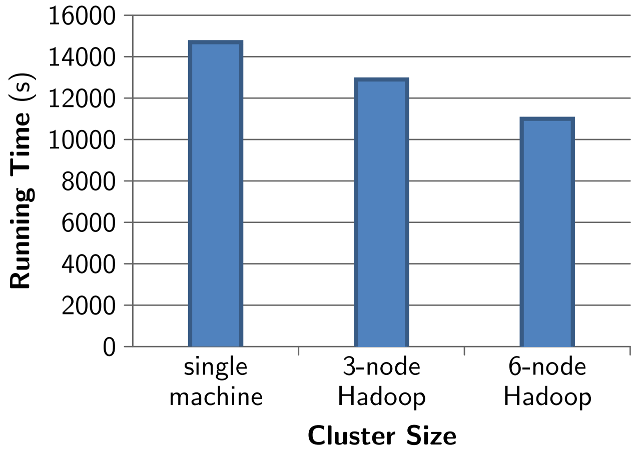


Figure 5.4. MD5 computation verification time

large objects on its stack at checkpoint times. Accordingly, an optimal implementation should select checkpoint positions strategically so as to avoid such overhead when possible.

Figure 5.5 reports certification times for SHA-1 computations over various input string lengths using 6 checkpoints. For our small cluster, 6 checkpoints was an optimal number (more and fewer checkpoints yielded higher overall runtimes). We expect that with larger clusters the optimal number of checkpoints rises proportionately to the cluster size. With a single machine, the verification time climbs rapidly with the input string length, whereas for Hadoop clusters it climbs much more slowly. For instance, where the performance difference between 3-node Hadoop and a single machine is about 5.5% for a 38KB string, it is about

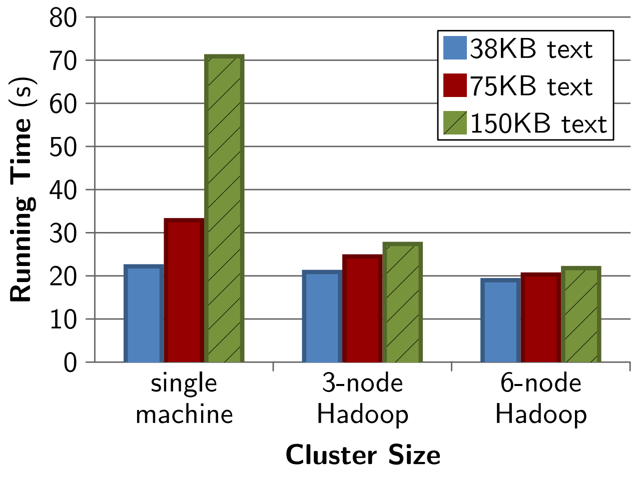


Figure 5.5. SHA-1 verification times with 6 checkpoints

60% for a 150KB input (4 times as large). Thus, the longer the serial computation, the greater advantage is observed for the parallelized validation architecture.

CloudCover's parallelized verification strategy scales best when applied to certify computations whose time-complexities are greater than their space-complexities. In these cases, CloudCover's division of the original computation's runtime across n nodes introduces speedups that rapidly outpace the overhead introduced by checkpoint operations, which typically have time complexity equal to the computation's space-complexity. To illustrate, Figure 5.6 compares the certification times for a quadratic-time, linear-space, insertion sort computation against the runtimes of the original, serial computation without any checkpointing. With a 600K-element input array and a 6-node cluster, certification completes 74% faster than the original computation on the same cloud. (Since the original computation is serial, the cloud cannot parallelize it and it runs on only one node.)

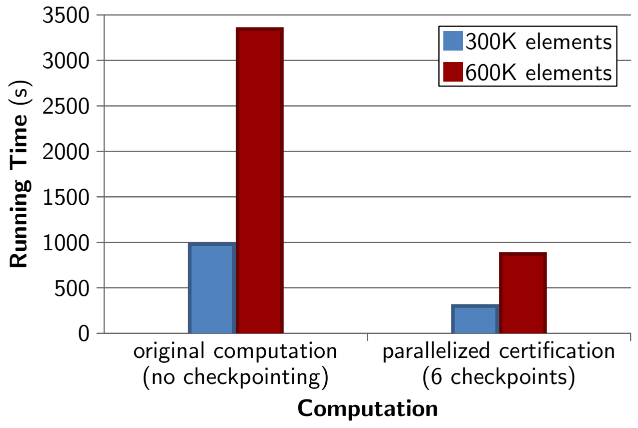


Figure 5.6. Original computation runtimes vs. verification runtimes for insertion sort on a 6-node Hadoop MapReduce cluster

CHAPTER 6

SILVER LINING: ENFORCING SECURE INFORMATION FLOW AT THE CLOUD EDGE[1]

Cloud computing has attracted tremendous attention over the past several years as a means to shrink IT expenditures, improve scalability and reduce administration overhead. As a result, cloud computing platforms (e.g., Amazon, 2013; Microsoft, 2013; Google, 2013a; Apache, 2013) have become extremely popular and extensively used. Both government and industry are adopting new business practices to maximize their effectiveness. The General Service Administration (GSA) recently announced they have accomplished a cost savings of almost $2 million USD per year since migrating from Lotus Notes to Google's cloud-based email (Coleman, 2012). According to Gartner, the typical IT organization invests two-thirds of its budget in daily operations, but moving to the cloud is expected to free up 35–50% of operational and infrastructure resources (Wilcox, 2010).

The ongoing mass shift to clouds for large-scale computing has raised significant concerns about security, however. More than 50% of global 1000 companies are projected to store sensitive data in public clouds by 2016 (Smith et al., 2011). It is therefore not surprising that many customers and businesses have become worried about security and privacy issues in the cloud. To address these concerns, the last few years have seen extensive research on adding greater security to cloud platforms. Examples include ensuring cloud computation integrity (e.g., Chow et al., 2009; Santos et al., 2009; Khan and Hamlen, 2012b), protecting cloud customer privacy (e.g., Takabi et al., 2010; Pearson et al., 2009; Khan and Hamlen,

[1] Submitted to the *IEEE International Conference on Cloud Engineering (IC2E)*, 2014. Safwan M. Khan, Kevin W. Hamlen and Murat Kantarcioglu. Silver Lining: Enforcing Secure Information Flow at the Cloud Edge. (submitted)

2012a), secure data storage in the cloud (e.g., Nepal et al., 2011; Bowers et al., 2009), and detection and prevention of software tampering (e.g., Fukushima et al., 2011; Khan and Hamlen, 2013a).

A common challenge faced by many of these efforts is the *multitenant* problem (Vaquero et al., 2011). In the complex and distributed environment of clouds, many users have simultaneous access to shared computing resources. This invites attacks that corrupt, deny, or infer secret details of computations or data owned by others. Many security fears associated with cloud computing therefore revolve around incomplete isolation of these myriad users. A prominent example is the debate over cloud computing for healthcare data management (Ponemon Institute, 2013). Healthcare data are often sensitive for a human lifetime or longer, and their disclosure are governed by elaborate regulations in many countries of the world (e.g., U.S. Department of Health & Human Services, 2013). While data encryption is a widely used protection while such data is at rest, it does not suffice to protect the data while it is decrypted for use in computations.

A large category of cloud security research has therefore concerned the enforcement of various forms of data access control in clouds. Most of these protections are implemented by modifying the cloud infrastructure (e.g., Bellessa et al., 2011; Bacon et al., 2010) or the underlying OS (e.g., Roy et al., 2010). Others add an extra access control layer atop an existing architecture, requiring new protocols (e.g., Ruj et al., 2011; Popa et al., 2010). While all of these provide effective enforcement, they have the significant drawback of being difficult to maintain as the cloud infrastructure evolves. Clouds are an extremely dynamic technology; there are constant improvements being made to enhance the efficiency of job dispatch, file lookup, data sharing, and a host of other implementation details. Security systems that customize the cloud implementation are often brittle to these version updates; they must be adapted or re-implemented frequently as the cloud evolves. This has been a significant deterrent to their adoption, and therefore a source of insecurity in real-world clouds.

To address this need, we propose a novel implementation approach, *SilverLine* (secure information flow verification in-lined), that enforces a large class of role-based access control and information flow security policies on untrusted cloud jobs, but whose implementation is completely separate and orthogonal to the rest of the cloud. This allows the cloud implementation and security implementation to be maintained fully independently, with changes to one having no impact on the other. Our approach realizes the policy enforcement as an *In-lined Reference Monitor* (IRM) (Schneider, 2000) whose programming is in-lined into untrusted binary jobs as they arrive at the cloud edge. After in-lining, the modified jobs *self-enforce* the security policy. Thus, no additional security monitoring within the cloud is needed.

To illustrate one large class of policies that can be elegantly enforced using this strategy, SilverLine enforces Mandatory Role-Based Access Control (MAC RBAC) policies that restrict explicit information flows between cloud users, roles, jobs, and resources (e.g., files). The in-lined enforcement code maintains and consults an *information flow graph* (IFG) implemented as a distributed data resource within the cloud. The IFG tracks information flows between the various principals, and the IRM prohibits job operations that introduce explicit flows that violate an administrator-defined policy. The IRM approach is well established for enforcing many important data confidentiality and integrity policies (Schneider, 2000; Bauer et al., 2005; Hamlen et al., 2006b,a; Ligatti, 2006; Hamlen, 2006; Hamlen and Jones, 2008; Aktug et al., 2008; Ligatti et al., 2009; Hamlen et al., 2012), but has not yet seen adoption in clouds.

SilverLine leverages *Aspect-Oriented Programming* (AOP) (Kiczales et al., 1997) to elegantly specify, implement, and in-line IRMs into untrusted jobs without access to job source codes. A *rewriter* automatically transforms untrusted jobs (Java bytecode binaries) via *aspect-weaving* as a preprocessing step before passing them to the cloud. To our knowledge, SilverLine is the first work that adopts IRMs in clouds to in-line information flow enforcement into jobs. It yields rewritten, self-monitoring cloud jobs without modifying the cloud platforms. These features establish it as an exceptionally practical and portable framework for adding powerful, custom security features to commodity clouds.

To evaluate our system, we deploy it in a realistic cloud environment—the popular Hadoop MapReduce (Apache, 2013). IRMs are implemented as AspectJ (Kiczales et al., 2001) *pointcuts* and *advice*. In AOP, a pointcut is a program element that identifies *join points* (binary program operations), and exposes data from the execution contexts of these join points to advice code that modifies or replaces them. The advice can thereby implement a policy that constrains all operations matched by the pointcut. Together, the pointcuts and advice form an *aspect*. AOP has been heralded in the software engineering community as a means of implementing cross-cutting concerns, such as security and process auditing (e.g., logging). Our evaluation results demonstrate the efficiency and scalability of this approach to implementing cloud access controls.

In essence, SilverLine decentralizes the trust on the user side. We augment the security and move the trust to the checker instead of the job submitters or cloud itself.

6.1 System Overview

6.1.1 Cloud Structure

Clouds typically provide at least three common categories of services to customers:

- *Software-as-a-Service (SaaS)* models provide software to users who do not wish to manage the network, servers, software, OS, or storage. Users consume the software from the clouds. Examples include Salesforce.com (SalesForce.com, 2013) and Google Apps (Google, 2013b).

- *Platform-as-a-Service (PaaS)* models provide users the facility to deploy their software on clouds. Users control their own software, but do not manage network, servers, OS or storage. Examples include Cloud Foundry (VMWare, 2013) and Google App Engine (Google, 2013a).

- *Infrastructure-as-a-Service (IaaS)* models benefit users with access to the infrastructure of the cloud to deploy their own resources. However, users do not manage the infrastructure. Examples include Amazon EC2 (Amazon, 2013) and Windows Azure (Microsoft, 2013).

In all of these types of services, users have varying degrees of access to security-sensitive cloud resources, based on the services they consume. To offer the broadest possible support, SilverLine remains mostly agnostic to the specific services offered by the underlying cloud. Different cloud providers may vary in the details of their internal architectures (cf., Apache, 2013; Amazon, 2013; Microsoft, 2013), though in general we assume that at some level their topologies consist of a few *master nodes* and a large collection (e.g., hundreds of thousands) of *slave nodes*. The complexity of the data dependencies and jobs that manipulate such vast resources invite subtle errors or malicious attacks that might violate users' information flow requirements, motivating a strong, flexible approach to enforcing such policies.

We deploy our system on Hadoop MapReduce (Apache, 2013), which consists of a single *NameNode* (with backups) as master node and a large number of *DataNodes* as slave nodes. The NameNode manages the HDFS namespace and regulates access to files by users. Additionally, it dispatches and monitors the computation all over the cloud cluster. DataNodes are typically distributed one per node in the cluster, managing storage attached to the nodes on which they run, and independently executing the user-submitted job fragments they are allocated. Users communicate with the NameNode, which coordinates the services from the DataNodes.

6.1.2 Access and Information Flow Control

SilverLine facilitates enforcement of mandatory information flow policies that constrain the explicit flow of information from one security principal (*viz.*, user, role, job, or resource) to another along non-side channels within the cloud. We chose this class of policies because it is simple to understand, arises often within multitenant, distributed workflows, has a well-established theory of enforcement, and yet is not supported by most commodity workflow clouds. Thus, a facility to add such support to existing clouds without any modification to the cloud implementation or infrastructure demonstrates the advantages of our approach.

As an example, a user U who owns a confidential file H might wish to grant limited read-access to H with some assurance that careless, faulty, or malicious readers of H do not subsequently leak that data to a world-readable file L. One way to enforce such a policy is to prohibit readers of H from subsequently writing to L. However, that simple enforcement strategy does not address *information laundering* scenarios, in which a principal P_1 copies H to an intermediate file I, after which another principal P_2 copies I to L. Sound enforcement of information flow policies therefore requires maintaining a transitive relation between data sinks and sources.

SELinux (National Security Agency (NSA), 2013) uses such a strategy to enforce role-based access control policies that constrain explicit information flows. Because of their broad usefulness, past work has integrated support for SELinux policies into Hadoop, but at the cost of non-trivial modifications to the cloud implementation (Roy et al., 2010). SilverLine does so without modifying the cloud.

6.1.3 Threat Model

Our threat model only concerns explicit, inter-principal, information flows within the cloud. In particular, it does not concern implicit information flows (e.g., flows that subtly divulge information by *not* exhibiting an otherwise observable action), or side-channels (e.g., where the attacker observes the timing or power consumption of jobs to infer secrets). We also do not secure flows outside the cloud. For example, flows that involve one malicious user communicating a secret to another outside the cloud, who then discloses it publicly within the cloud, are not secured by SilverLine. All of these forms of confidentiality violation are important ongoing subjects of extensive study (cf., Sabelfeld and Myers, 2003), but are outside the scope of our present investigation.

We conservatively assume that jobs submitted to the cloud might contain arbitrary malicious programming, and that attackers know all details of SilverLine's enforcement strategy and implementation. For example, attackers might submit jobs containing malicious code that anticipates the IRM logic that will be in-lined by SilverLine, and that seeks to destroy or circumvent it at runtime to violate the policy. Thus, SilverLine's security is not based on obscurity; knowledge of its implementation does not facilitate successful attacks. The cloud kernel, Java Virtual Machine (JVM), and underlying OS and hardware are all trusted. In particular, we assume jobs come in the form of syntactically valid, type-safe Java bytecode binaries, since malformed or type-unsafe binaries are automatically rejected by the trusted JVM.

Security-relevant job operations mainly consist of system API calls, which are the only means for Java code to perform explicit I/O. There are three main APIs of interest: (1) the HDFS API, (2) Java's standard runtime API, and (3) the API exposed by the OS to user processes. The last of these is only directly exposed to Java programs via Java's native code interface, which is unsafe and should not be used by jobs compiled for Hadoop. We therefore adopt the standard precaution of denying native code privileges to all jobs.

Java's runtime API contains many unsafe I/O operations that (disturbingly) remain fully available to unprivileged jobs in a standard Hadoop installation. Since the HDFS files are stored in file objects maintained by the OS, and since the Java I/O libraries afford direct, uncensored access to this OS view, we found that it is trivial to write malicious Hadoop jobs that abuse the Java runtime to bypass all the HDFS access controls. SilverLine closes this vulnerability by prohibiting access to the Java runtime's I/O methods; jobs may only access files via HDFS. The remainder of the IRM implementation focuses on guarding HDFS API calls.

6.1.4 MapReduce Paradigm

MapReduce (Dean and Ghemawat, 2008) is an increasingly popular distributed programming paradigm used in clouds. It provides automatic parallelization and distribution, fault tolerance, I/O scheduling, monitoring, and status updates among other features, making it popular among the commonly used cloud platforms (e.g., Apache, 2013; Amazon, 2013). Since our system is deployed on Hadoop, we leverage Hadoop's MapReduce framework.

In Hadoop, user computations arrive as Java bytecode programs (jobs) submitted by users. Each job consists mainly of two functions: *Map* and *Reduce*. The Map function maps input key-value pairs to a set of intermediate key-value pairs. Based on the configuration, those may be passed to *Shuffle* or *Sort* functions for additional processing. The Reduce function reduces the set of intermediate key-value pairs that share a key to a smaller set of key-value pairs traversable by an iterator. Each Map process and Reduce process works independently on DataNodes without communication.

6.1.5 Policy Specifications and Flow Tracking

An administrator-specified *policy file*, such as the one in Figure 6.1, defines the specific access control and information flow policy enforced by SilverLine. Command "mayaccess P_1 P_2" grants P_1 direct write-access to P_2. If P_1 is an object (e.g., file) and P_2 is a subject (*viz.*, user, role, or job), then it grants P_2 read-access to P_1. (From a policy standpoint, it is as if file P_1 "wrote" to user P_2.) Rule "noflow P_1 P_2" disallows information flows from P_1 to P_2.

```
mayaccess role1 file2;
mayaccess file4 role1;
mayaccess role2 file6;
mayaccess role2 file9;
mayaccess file12 role2;
mayaccess file1 role3;
mayaccess file3 role3;
mayaccess role3 file7;
mayaccess file8 role4;
mayaccess role4 file10;
mayaccess file11 role5;
mayaccess file13 role5;
mayaccess role5 file15;
mayaccess role5 file17;
noflow file4 file2;
noflow role1 file14;
noflow file12 role1;
noflow role1 file15;
noflow file8 role3;
noflow file18 role4;
noflow file20 file16;
noflow file5 file19;
```

Figure 6.1. Sample Policy File

For example, in Figure 6.1, even though role1 may read file4 and write to file2, it may not do the former followed by the latter, since flows from file4 to file2 are disallowed.

The policy language in Figure 6.1 is simplistic but suffices for our experiments. It can be conceptualized as a classic Bell-LaPadula labeling system (Bell and Lapadula, 1973) where the labels form a lattice of principal subsets (expressing impermissible flow destinations) ordered by subset relation (Denning, 1976; Sandhu, 1993). Much more expressive policy languages (e.g., Swamy et al., 2008; Hamlen and Jones, 2008) are possible, but supporting them is a subject of future work.

To enforce the policy, SilverLine maintains an *Information Flow Graph (IFG)* stored as a distributed data structure in HDFS. Figure 6.2 depicts an example IFG. Nodes are labeled with principals (*viz.*, roles, users, jobs, and resource identifiers) as well as the set of nodes to which their information *must-not* flow (shown as dashed edges in the figure). For example, to enforce the policy in Figure 6.1, node role1 is labeled with must-not set $MN[\mathsf{role1}] = \{\mathsf{file14}, \mathsf{file15}\}$. Solid, directed edges indicate active information flows (e.g., files currently opened by running jobs) between the nodes.

Whenever a new edge (x, y) is about to be introduced, SilverLine checks whether $MN[x] \cap R(y) = \emptyset$, where $R(y)$ is the set of nodes reachable from y. If not, the impending operation is rejected by throwing a (catchable) exception. Otherwise, the operation is permitted and $MN[x]$ is propagated to the nodes in $R(y)$. The bold red edges in Figure 6.2 show such a

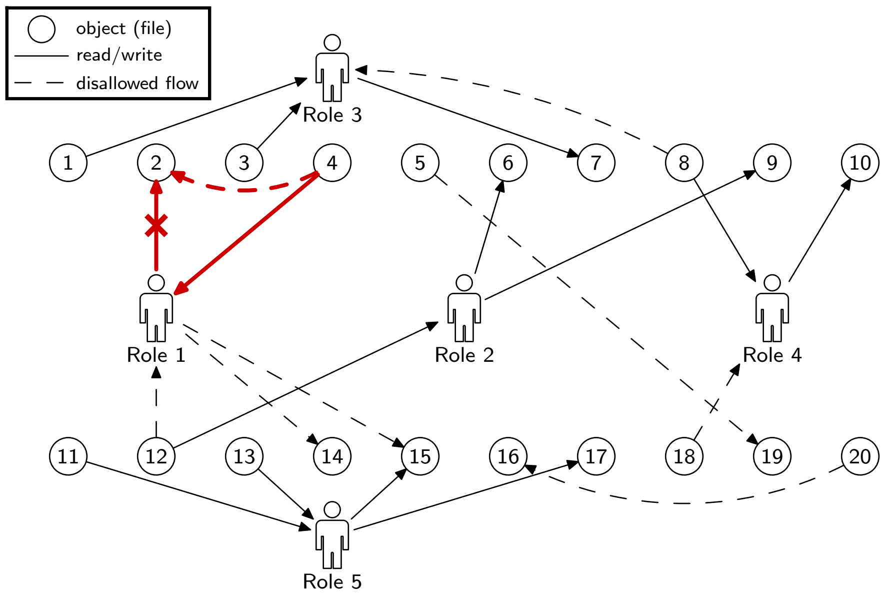


Figure 6.2. A sample Information Flow Graph (IFG). Edge (role1, file2) is rejected because its addition would complete disallowed flow (file4, file2).

policy violation. The addition of edge (role1, file2) to the graph would complete disallowed flow (file4, file2), so role1's request to write to file2 is rejected.

To avoid a bottleneck when accessing the IFG, it is stored as a distributed HDFS data structure whose disconnected components can be separately locked for exclusive write-access. Since the IFG only includes edges for current operations, and since reading and writing to the same file simultaneously is rare (it risks inconsistent results in HDFS), the IFG is usually quite fragmented; having separate locks for disconnected components therefore scales well. HDFS does not provide built-in locking support, so we implemented it ourselves as part of the IRM's access protocol (without modifying the cloud). All access to the IFG data by the non-IRM job code is prohibited by the IRM.

6.1.6 In-lined Reference Monitor Design

SilverLine's main implementation is an AspectJ (Kiczales et al., 2001) program that encodes the transformation of untrusted job code into safe job code as a set of aspects. Each aspect's pointcuts identify unsafe program operations (API calls) that might appear in untrusted jobs, and its advice supplies guard code that secures such operations wherever they appear. The guard code includes logic for consulting and updating the IFG at runtime. AspectJ's aspect-weaver inserts the guard code around each unsafe operation prior to running the job, resulting in a secure binary.

In contrast to purely static approaches to mobile code security, SilverLine makes no attempt to decide in advance whether untrusted jobs, when executed, might try to violate the policy. (In fact, information flow policies are statically undecidable in general (Hamlen et al., 2006b).) Rather, it introduces programming that discovers impending policy violations at runtime and intervenes to prevent them.

Protecting the integrity of the IRM from corruption by the surrounding untrusted job code is a major part of the design. There are three major ways that malicious jobs might attack the IRM, none of which are successful:

1. Malicious jobs might try to erase or overwrite the IRM code at runtime.
2. Malicious jobs might try to corrupt the IRM's internal variables or data.
3. Malicious jobs might try to "jump over" the IRM's runtime security checks to reach policy-violating operations, bypassing the security code.

Attack 1 is thwarted by denying jobs access to Java's reflection libraries, which are the only way to write self-modifying code in Java. Attack 2 is thwarted by storing all the IRM's data in local variables and private field members of a non-inheritable class. Type-safety of the Java bytecode language therefore prevents untrusted job code outside of that class from corrupting those members. Finally, attack 3 is not possible because the aspect-weaver retargets all static control-flow transfer instructions so that they cannot bypass the advice. The only form of dynamic control-flow in Java bytecode is method call, which can only target

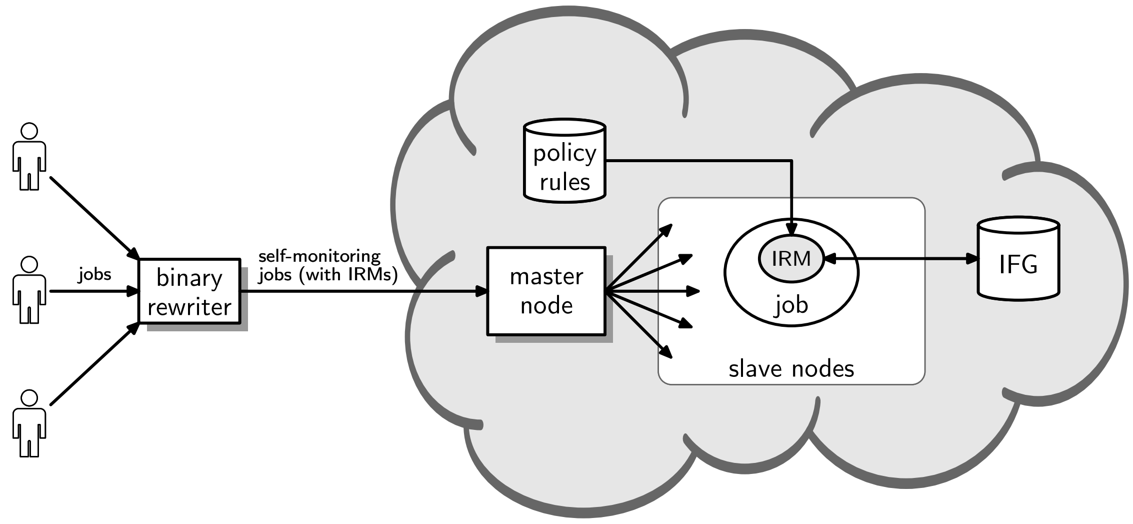


Figure 6.3. System architecture of SilverLine

a method entrypoint, and there are no method entrypoints between the security checks and the dangerous operations they check. These approaches to ensuring IRM integrity have been validated by prior studies (Hamlen et al., 2006a; Aktug et al., 2008; Hamlen et al., 2012).

6.1.7 Architecture and Protocol

Figure 6.3 depicts the high-level architecture of SilverLine. End users submit jobs to the cloud in the usual way; no change to how jobs are created or submitted is required to accommodate SilverLine. SilverLine's aspect-weaver intercepts the submitted jobs at the cloud edge and in-lines the IRM. The aspect-weaver may reside on any node inside cloud, or may be deployed on a separate machine outside of cloud. The resulting self-monitoring binaries are then dispatched to the cloud for execution.

SilverLine maintains persistent access history. For example, if user1 runs a job that reads from file4, and then later runs a separate job that writes to file2, the IFG maintains the node labels resulting from the earlier job and thereby detects the flow from file4 to file2.

There are interesting design challenges for handling this correctly in a cloud where jobs run concurrently and job submissions are separate from job executions. For example, if

Algorithm 7 Initialize IFG

```
1: for (noallow $x$ $y$) $\in$ $policy$ do
2:     $MN[x] \leftarrow MN[x] \cup \{y\}$
3: end for
```

user1 first submits job1 and job2 in their entirety to the cloud, and then job1 and job2 run concurrently, with job1 only reading from file4 and job2 only writing to file2 (no other file accesses or network accesses), then SilverLine concludes that no information has flowed from file4 to file2 (yet). The reasoning is that job2 was submitted to the cloud (i.e., written) before job1 read file4, and the two jobs did not (explicitly) communicate. Such scenarios are why SilverLine needs separate nodes for users, roles, and jobs.

The IFG initialization and IRM runtime guard code are summarized in Algorithms 7 and 8, respectively. Line 16 of Algorithm 8 computes the IFG subset reachable from y, which is typically small due to the IFG's disconnectedness (see Section 6.1.5). Line 20 expresses IFGs as multisets, since duplicate edges can arise from multiply opened file handles.

6.2 Implementation

Implementation of SilverLine in a real-world cloud environment is one of our main contributions. We deployed it on a commodity data processing cloud—Hadoop MapReduce (Apache, 2013). Experiments were conducted on a Hadoop cluster consisting of 12 DataNodes and 1 MasterNode. Nodes have Intel Pentium IV 2.40–3.00GHz processors with 2–4GB of memory each, running Ubuntu operating systems.

As mentioned in Section 6.1.6, SilverLine implements IRMs using AspectJ (Kiczales et al., 2001). Listing 6.1 shows a (simplified) sample aspect that implements part of Algorithm 8.

For the experiments, the SecurityExceptions that signal policy violations are not caught by the surrounding job code, so the job simply aborts. Jobs could alternatively take corrective actions, such as handling the exception by rolling back to a consistent state (but corrective

Algorithm 8 SilverLine Guard Pseudo-code for Untrusted Operation *op* on HDFS Object *o*

```
1: if $op$ is a Java I/O API call then
2:   throw SecurityException
3: else if $op$ is an HDFS open/close operation then
4:   if $op$ open/closes $o$ for reading then
5:     $x \leftarrow o$
6:     $y \leftarrow job\_identifier$
7:   else                                        // $op$ open/closes $o$ for writing
8:     $x \leftarrow job\_identifier$
9:     $y \leftarrow o$
10:  end if
11:  if $op$ is a close then
12:    $IFG \leftarrow IFG - \{(x, y)\}$
13:  else if (mayaccess $x$ $y$) $\notin policy$ then
14:    throw SecurityException
15:  else if $(x, y) \notin IFG$ then
16:    $R \leftarrow breadth\_first\_search(IFG, y)$            // $R$ is small
17:    if $R \cap MN[x] \neq \emptyset$ then
18:      throw SecurityException
19:    else
20:      $IFG \leftarrow IFG \uplus \{(x, y)\}$
21:      for $z \in R$ do
22:        $MN[z] \leftarrow MN[z] \cup MN[x]$
23:      end for
24:    end if
25:  end if
26: end if
```

actions cannot violate the policy, since the code that implements any corrective action is part of the job and therefore constrained by the IRM). We distribute AspectJ JARs and aspects to all the nodes in the cloud to enable it in the Hadoop environment. SilverLine takes advantage of AspectJ's *runtime weaving* feature that injects specified security policies into the binaries of jobs instead of into source code at compile or post-compile time.

To maintain persistent history and track concurrent jobs (Section 6.1.7), jobs are represented in IFGs as temporary nodes. When a job J is received from a user U, we create a new node for J and copy $MN[U]$ to $MN[J]$. This reflects the fact that J potentially knows all

```
public aspect CloudIRM {

  pointcut reads(InputStream f) :
    call(* * org.apache.hadoop.fs.FileSystem.open(..) && args(f,..);

  // Guard HDFS I/O calls
  before(InputStream f) throws SecurityException : reads(f)
  {
    if (!policyMayAccess(f, this_job))
      throw new SecurityException("access denied");
    lockIFGComponents(IFG, f, this_job);
    try {
      if (!hasEdge(IFG, f, this_job)) {
        Collection⟨IFGNode⟩ r = BFS(IFG, this_job);
        Collection⟨IFGNode⟩ mn = getLabel(IFG, f);
        for(Iterator⟨IFGNode⟩ i=r.iterator(); i.hasNext(); ) {
          if (mn.contains(i.next()))
            throw new SecurityException("info flow violation");
        }
        addEdge(IFG, f, this_job);
        for(Iterator⟨IFGNode⟩ i=r.iterator(); i.hasNext(); ) {
          IFGNode v = i.next();
          setLabel(IFG, v, getLabel(IFG,v).addAll(mn));
        }
      }
    } finally { unlockIFGComponents(IFG, f, this_job); }
  }

  // Prohibit Java I/O calls
  before(..) throws SecurityException : call(* * java.io.*(..)) {
    throw new SecurityException("prohibited operation");
  }

}
```

Listing 6.1. Sample aspect in AspectJ

secrets known by U at the time U submitted J. Whenever J opens a file F_1 for reading, we join (union) (Sandhu, 1993) $MN[F_1]$ into $MN[J]$ (and all nodes reachable from J) to reflect the flow of secrets from F_1 to J. Dually, whenever J opens F_2 for writing, we join $MN[J]$ into $MN[F_2]$ (and all nodes reachable from F_2). When J finishes and its results are returned to U, $MN[J]$ is joined back into $MN[U]$; then we destroy node J and all its adjacent edges. For efficiently controlling concurrent access to our IFG in HDFS, we implement a straightforward synchronization mechanism using semaphores. Far more sophisticated distributed

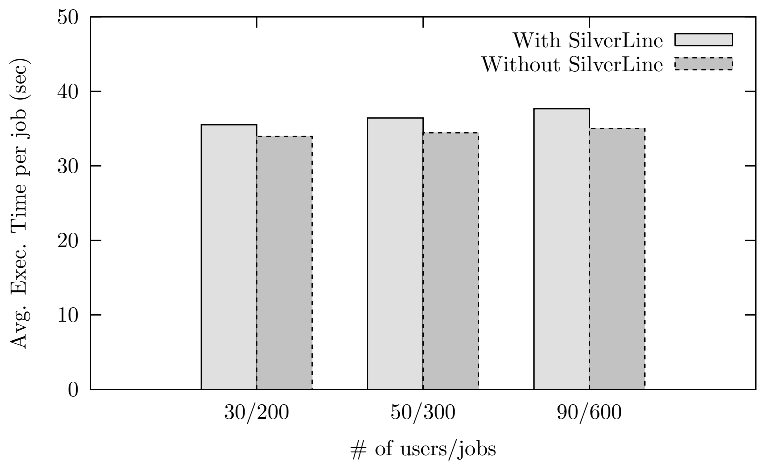


Figure 6.4. Performance measurement with increasing concurrency

graph representations are possible (cf., Mondal and Deshpande, 2012), but are left as future work.

6.3 Experimental Results

To evaluate our system, we conduct two experiments: one with synthetic jobs and one with real-world jobs drawn from public MapReduce code repositories. All experiments use 5 roles and include attacks (Section 6.1.3) that attempt to violate the information flow policies or subvert the IRM. SilverLine successfully blocks all these attacks in our experiments by halting such jobs before their first violating operations. The remainder of this section reports performance results for the remaining policy-compliant jobs that were not prematurely terminated.

The first experiment runs *chaotic jobs* whose Mappers perform randomly chosen HDFS file operations in random combinations, as well as dangerous Java I/O and system calls (via `java.io.Runtime`) with some probability. Figure 6.4 reports performance results, which illustrate the good scalability of SilverLine. Each trial simulates increasing numbers of users and jobs simultaneously, taxing the system's scheduler and shared resources (e.g., the IFG). Under these conditions of high resource contention, jobs with SilverLine's IRM installed run

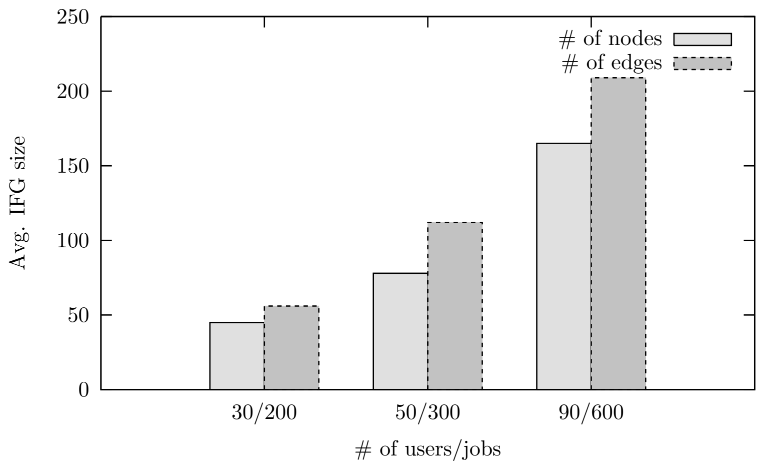


Figure 6.5. Performance measurement with increasing IFG size

4.6–7.5% slower (approximately 1.56–2.64 seconds slower) each. This slowdown is primarily due to locking and unlocking of the distributed IFG data structure for writing, and by the extra job operations that implement the IRM in each job. Since each lock is acquired for only a very short time (see Listing 6.1), the overhead remains reasonable.

Figure 6.5 reports the IFG size for the same experiment. The number of IFG nodes increases with the increasing number of job submissions over the trials, while the edge count increases due to the higher number of concurrently accessed HDFS file objects. The median IFG sizes (nodes plus edges) range from 101–374 over the three trials. This means that each job contributes just 0.6 nodes+edges to the average size of the IFG. This excellent scalability is because the IFG only tracks concurrent resource accesses, and HDFS avoids longstanding locks on files. The resulting median job performance overhead comes to a mere 66ms total extra job time per IFG node or edge. This low impact is due to HDFS's aggressive distribution of the IFG over many cloud nodes, and the generally small number of nodes reachable from any given node (since long edge chains only result from separate jobs simultaneously reading *and* writing the same file—usually in error).

Our second experiment (Figure 6.6) examines performance under more practical conditions that involve some common, pre-existing MapReduce programs used in the field. We

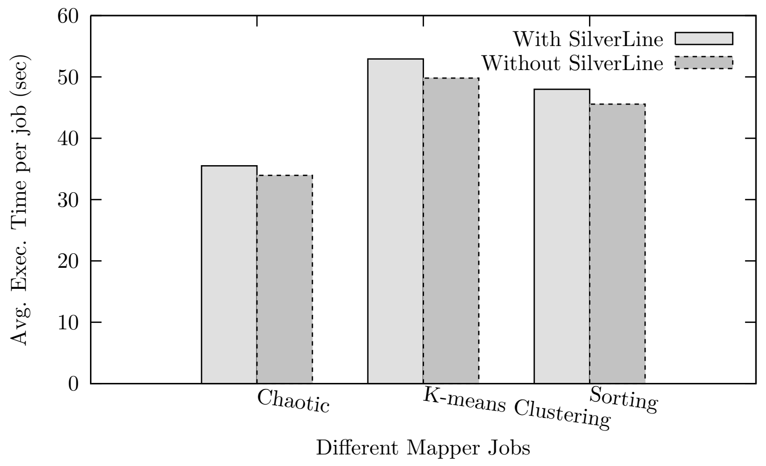


Figure 6.6. Comparing performance of different MapReduce jobs

choose two classic MapReduce jobs: (1) *k-means clustering*, which partitions data points into 2 clusters, and (2) *sorting*, which sorts data provided by the input files using a standard merge-sort algorithm. These were submitted with randomly generated input files to Hadoop as 200 jobs from 30 simulated users. The experiments were performed both with and without SilverLine (with the results shown as pairs of bars in the figure) in order to assess SilverLine's performance overhead relative to standard Hadoop. We also applied the same experiment to the chaotic jobs.

Figure 6.6 shows that SilverLine introduces 4.39%, 5.88% and 5.04% performance overhead, respectively, for chaotic, k-means clustering, and sorting jobs, respectively. The experiment also demonstrates the applicability of our system to existing cloud job codes. No change to the job binaries was required; they worked seamlessly on Hadoop after instrumentation by SilverLine.

CHAPTER 7

RELATED WORK

Cloud computing has received significant attention from research communities in academia as well as industry; however, there are many challenges facing cloud computing to be widely deployed and used. The major, and increasingly demanding, challenge is security.

Cloud computing security has exploded into a vast research area in recent years. There have been lot of issues and corresponding research work, which are demonstrated in many articles, including those by Hamlen et al. (2010); Kulkarni et al. (2012); Xiao and Xiao (2012); Dahbur et al. (2011); Bhadauria and Sanyal (2012). It is beyond the scope of this dissertation to discuss all of them. We rather describe the works below related to our contributions.

7.1 Hatman

Much of the work in cloud computing security focuses on data privacy (Ryan, 2011). Data integrity is a second major concern that involves challenges related to secure storage (e.g., Nepal et al., 2011; Bowers et al., 2009) and secure integrity attestation of computation results. The latter is the subject of our Hatman.

AdapTest (Du et al., 2011) and RunTest (Du et al., 2010) implement cloud service integrity attestation for the IBM System S stream processing system (Gedik et al., 2008) using attestation graphs. Always-agreeing nodes form a clique in the graph, facilitating detection of malicious collectives.

In contrast, our work considers a reputation-based trust management approach to integrity violation detection in Hadoop clouds. Trust management systems probabilistically anticipate future misbehavior of untrusted agents based on their histories of past behavior.

Reputation-based trust managers, such as EigenTrust (Kamvar et al., 2003), NICE (Lee et al., 2003), and DCRC/CORC (Gupta et al., 2003), assess trust based on reputations gathered through personal or indirect agent experiences and feedback.

(Abawajy, 2009) claims to take the first attempt of trustworthiness in inter-clouds computing environment. It uses peer-to-peer concept for their fully distributed approach where peers are the master nodes from each cloud. However each cloud does not connect with all other clouds in the inter-clouds, rather they connect to zero or more clouds based on some policy. Reputation calculation is inefficient as a component named Reputation Manager is introduced in each cloud to do that task. It is not clear how the feedback is provided from one peer to another. It does not deal with malicious resources explicitly and does not come up with any experimental results. (Li and Ping, 2009) is for inter-clouds environment and is distributed as well. However, its trust management system's trust calculation is ambiguous and not based on feedback. It includes no real time experiments, simply simulation results.

(Manuel et al., 2009) builds trust management system for intra-cloud environment and is centralized. For trust management system, it employs user's feedback and other factors which are predefined to calculate trust values - consequently it is a policy-based trust management system. The procedure is not efficient as master node does all the calculation for each data node in the cloud. It presents some simulation results but does not implement it on a real cloud platform. TrustCloud (Ko et al., 2011) proposes a trusted cloud framework which addresses accountability in cloud computing via technical and policy-based approaches emphasizing on accountability rather than privacy component of trust we deal in this research work.

Opera (Nguyen and Weisong, 2010) employs reputation-based trust management to improve Hadoop computation efficiency. It tracks node trust as a vector of efficiency-related considerations, such as node downtime and failure frequency. However, malicious behavior in the form of falsified computation results are not considered, making it unsuitable for protecting against data integrity attacks.

Policy-based trust management (Blaze et al., 1998) has been used as a basis for allowing cloud users to intelligently select reliable cloud resources for their computations, and to provide accountability of cloud providers to their customers (Manuel et al., 2009; Ko et al., 2011). These approaches necessarily involve re-architecting clouds to expose some or all of their internal resources to users, so that users can make informed choices regarding those resources.

Peer-to-peer, distributed, decentralized trust management has also been recognized as a natural means of providing inter-cloud security guarantees (Abawajy, 2009; Li and Ping, 2009). Each cloud acts as an individual peer in a super-cloud with no central authority. Inter-cloud computations are then partitioned and distributed based in part on cloud reputations.

As an alternative to trust management, traditional byzantine fault tolerance has been used extensively to detect and isolate malicious behavior in networks of replicated services, including clouds (Zhang et al., 2011; Kotla et al., 2009). However, these solutions typically involve implementation of new communication protocols for untrusted agents, complicating their application to existing, large-scale cloud implementations such as Hadoop.

Although there is strong evidence that EigenTrust and similar trust management approaches scale well to networks with large numbers of data nodes (West et al., 2010), NameNode scalability is not something we studied. Hadoop's use of a single NameNode has been identified in the literature as a potential bottleneck (Shvachko, 2010), and several current works are exploring the feasibility of distributing NameNode computations and metadata (Molina-Estolano et al., 2010; Talwalkar, 2011). Hatman benefits from these advancements, since they offer opportunities to more widely distribute its trust matrix metadata.

7.2 AnonymousCloud

Data privacy concerns are widely recognized as a significant impediment to consumer confidence in cloud computing (Fujitsu, 2010; Ryan, 2011; Chen and Zhao, 2012). Associated challenges span at least three categories of related work: secure remote platform attestation (i.e., trusted computing), secure data storage, and information-centric security (Chow et al., 2009).

Trusted computing provides users high assurance that they are communicating with a remote server consisting of known, trusted hardware and software (Mitchell, 2005). Secure storage regards the problem of safely storing private data in the cloud (usually in encrypted form) between computations that use it (e.g., Huang et al., 2011). In contrast, information-centric approaches imbue data with self-protecting properties, such as by representing it in a form amenable to direct computation on cyphertexts without decryption (e.g., Liu et al., 2012). AnonymousCloud's approach of decoupling private data from its provenance information can be viewed as an instance of the last of these approaches.

General data anonymization is a vast research area spanning many decades; however, the most widely used strategies for anonymization of data content are currently differential privacy (Dwork, 2008) and k-anonymity for privacy-preserving microdata release (Samarati, 2001). Such research benefits our work by providing a means for customers to anonymize private data content before submitting it to the cloud. We therefore assume that customers interested in privacy submit data that divulges fewer secrets once it has been decoupled from provenance and semantic metadata, and that therefore benefits from our anonymization protocol.

Prior work has also explored decoupling document content from format and structure for more secure cloud storage and processing (Xu et al., 2009). For example, HTML documents can be encoded in a format that separates their tree structures from the textual content of elements and attributes. Since a majority of private data resides in the content, this allows separate processing of structural-based queries in the cloud without divulging the private data.

To decouple and conceal provenance metadata, AnonymousCloud employs onion routing based on Tor (Dingledine et al., 2004). Tor has become the most successful public anonymity communication service in the Internet, with tens of millions of users worldwide (Greenberg, 2012). In Tor, initiators choose a path through network and build a circuit in which each node or onion router in the path knows only its successor and predecessor, but no other nodes in the circuit. Based on the chosen path or route, the initiator first encrypts the data with one layer of encryption for each node in the path, from the last node to the first. This is likened to the layers of an onion, with each hop peeling one layer as the data is forwarded to its destination. The data can only be read in plaintext once it reaches the endpoint of the path and all layers have been peeled.

The Tor Cloud project (ExpressionTech and The Tor Project, 2012) has implemented a full-scale Tor system within a production-level cloud that runs on the Amazon EC2 cloud computing platform (Amazon, 2013). It provides a user-friendly way of deploying bridges to help users access an uncensored Internet. Tor Cloud conceals user pseudonyms (e.g., IP numbers) from untrusted third-party services, but does not suffice to anonymously access data from a third-party cloud (Laurikainen, 2010), since clouds require a means of authenticating users in order to control access to each user's private data and bill them appropriately.

Our work therefore extends cloud-based onion routing with an anonymous credential system for authentication (Chaum, 1985). Anonymous authentication provides zero-knowledge proof of identity, allowing data to be securely decoupled from provenance for enhanced privacy. More elaborate anonymous credential systems (e.g., Camenisch and Herreweghen, 2002; Zarandioon et al., 2011; Slamanig, 2011; Camenisch and Lysyanskaya, 2004; Jensen et al., 2010; Backes et al., 2005) support additional security properties, such as non-transferability, lazy revocation, and access hierarchies. These are not necessary for our system, but could be substituted if such properties are desirable for other reasons.

Our attack analysis and experiments do not consider the threat of end-to-end timing attacks (except that we mandate circuit lengths of at least 3 to preclude the simplest such attacks). Past works have shown that these attacks are potentially effective against Tor and other onion routing systems even when the attacker controls only a few nodes (Abbott et al., 2007; Hopper et al., 2010). The Tarzan system protects against timing attacks through generation of artificial cover traffic that masks timing patterns in a sea of mimicry and noise (Freedman et al., 2002). Future work should consider the feasibility of supplementing AnonymousCloud with similar protections.

Aside from implementing protections and protocols that directly facilitate greater privacy, mechanisms that provide greater transparency for internal cloud operations—particularly distribution and management of security-sensitive data—is critical for instilling greater confidence in end users (Abawajy, 2009; Ko et al., 2011; Nguyen and Weisong, 2010). Future work should therefore consider augmenting AnonymousCloud with features that afford customers greater control over data distribution and scheduling details after Tor circuit construction, and without sacrificing anonymity.

7.3 Penny

P2PRep (Cornelli et al., 2002) is one of the first works to implement secure reputation management in a real-world P2P network (Gnutella, 2010). Resource-requesting peers in the network assess the reliability of perspective providers before initiating downloads by polling large numbers of peers using broadcast messages. Poll responses are then aggregated by the requesting peer to estimate the desired integrity label or trust value along with trust values for all peers whose opinions were acquired by polling. This strategy has the advantage of being implementable atop the existing Gnutella network protocol, but it has the disadvantage that labels and trust values are not global and are not guaranteed to converge. That is, the integrity label or trust value obtained depend on which peers were polled, which in turn depends upon the poller's placement within the P2P network. Two peers at different locations in the network might therefore consistently derive different reputations for the same resource. Broadcast messages can also be expensive, requiring $O(b^d)$ messages to be sent, where b is the branching factor of the network and d is a time-to-live parameter dictating the maximum depth of the tree of peers being polled.

XRep (Damiani et al., 2002) enhances P2PRep by combining reputations of providers and resources, offering more informative polling and overcoming the limitations of strictly provider-based solutions. However, it still suffers the disadvantages of P2PRep above. ServiceTrust (He et al., 2009) is a service-oriented paradigm that computes trust globally, but at the cost of centralizing the system, inviting centralized points of failure. One solution is to compute trust globally using decentralized gossip-based algorithms (Bachrach et al., 2009). However, this does not recursively apply the trust system to the gossip itself, allowing malicious agents to potentially gain undue influence by reporting high trust for malicious allies.

In contrast to these unstructured approaches, Penny is implemented atop a structured P2P protocol—Chord (Stoica et al., 2001). Chord solves the fundamental problem of efficiently locating peers with particular data objects by assigning a unique identifier to each peer, and arranging them in a ring structure sorted by identifier. Each peer maintains a finger table of size m, where 2^m is the size of the identifier space. This enables peers to locate and contact the peer with a given identifier in $O(\log N)$ message hops, where N is the number of peers in the network. In Chord, each shared data object also has a single key-holder peer, who is charged with directing requesters of that object to peers that own it. To request an object, a peer can locate its key-holder in $O(\log N)$ message hops, whereupon the key-holder responds with a list of servers from which the object can be downloaded. Alternatives to Chord include CAN (Ratnasamy et al., 2001), Pastry (Rowstron and Druschel, 2001), Tapestry (Zhao et al., 2004), and MAAN (Cai et al., 2004). These systems offer distributed, scalable, and efficient search, but they do not include data security or privacy enforcement mechanisms. Penny extends Chord by providing a framework for maintaining centralized security labels for data shared over a Chord network.

In trust management systems, peers and occasionally objects in the system are labeled with trust values based on past peer interactions. These past experiences are consulted to predict future malicious behavior and incentivize good behavior. There are three major types of trust management systems. *Reputation-based* systems use knowledge of a peer's reputation (gathered through personal or indirect experience) to determine the trustworthiness of another peer. Examples include EigenTrust (Kamvar et al., 2003), DMRep (Aberer and Despotovic, 2001), P2Prep (Cornelli et al., 2002), XRep (Damiani et al., 2002), Sporas and Histos (Zacharia and Maes, 2000), PeerTrust (Xiong and Liu, 2004), NICE (Lee et al., 2003), and DCRC/CORC (Gupta et al., 2003). In contrast, *policy-based* trust management systems, such as PolicyMaker (Blaze et al., 1998), derive peer trust based on supplied credentials. Finally, trust management systems based on *social networks* determine trust by analyzing a complex social network. Examples include Marsh (Marsh, 1994), Regret (Sabater and Sierra, 2002), and NodeRanking (Rivest et al., 2001).

Penny integrates a reputation-based trust management system based on secure EigenTrust (Kamvar et al., 2003). Each peer is assigned a global trust value based on the peer's history of downloads. Global trust values are computed in a distributed manner with minimal load, resulting in assured convergence for all trust queries without centralization.

Resource Description Framework (RDF) is a metadata model for web data exchange. It is widely used for semantic web knowledge due to its expressive power, semantic interoperability, and reusability. We show that Penny is well-suited not only for traditional P2P file/object lookups/downloads, but also for deploying and querying RDF datasets. Two categories of prior work have investigated effective RDF data management in P2P environments. One considers the problem of distributing and retrieving RDF data efficiently (Newman et al., 2008; Cai et al., 2004; Newman et al., 2008), while the other proposes algorithms for efficient query processing (but not storage) (Verheijen, 2008; Liarou et al., 2007b,a). Neither body of work considers peer trust or data security issues to our knowledge.

Penny's RDF representation scheme is most closely related to that of RDFPeers (Cai et al., 2004). RDFPeers stores each RDF triple by hashing it three times (once for subject, predicate, and object, respectively), resulting in three replicas. Penny adopts a similar strategy, but for more efficient query processing it indexes each query by hashing only one of the three parts, retrieves a superset of the desired triples, and locally filters the results.

To ensure confidentiality, Penny peers must send some messages anonymously. This is accomplished via anonymizing tunnels (Chaum and van Heyst, 1991; Freedman et al., 2002; Zhu and Hu, 2008), which permit peers to route their messages through tunnels of randomly chosen peers. Multilayer encryption and randomly generated cover traffic prevent any peer in the tunnel from learning whether its successor is the message originator or just another hop in the tunnel. The tunnels are bidirectional, allowing recipients to reply without knowing the identity of the message originator. The approach has proved to be both flexible and scalable, requiring little overhead above that incurred by Chord's existing message-routing protocol (Chaum and van Heyst, 1991; Freedman et al., 2002).

7.4 CloudCover

Automatic result checking has been studied in the literature for at least a quarter century. It was first proposed as a means of debugging software (Blum and Kannan, 1989). Later work extended the idea to fault tolerance by observing that certain algorithms can be reformulated to yield a *certification trail* of data that witnesses the integrity of the algorithm's result (Sullivan et al., 1995). When available, such a trail can be verified independently by a distinct, faster certification algorithm to achieve efficient result checking. This insight has led to recent work in the formal methods community on frameworks for developing trail-producing software and their certifiers (Barthe et al., 2010).

Unfortunately, the addition of certification trails to software is non-trivial in general (justifying the application of formal methods). It typically requires reformulating and reimplementing the algorithm, as well as developing a completely new certification stage that is unique to each algorithm being checked. Applying the technique to most existing, production-level software is therefore a significant challenge.

Homomorphic encryption has been proposed as a way to cryptographically protect computation results on untrusted hosts, making incorrect results arbitrarily difficult for malicious hosts to forge (Sander and Tschudin, 1997). The disadvantage of these schemes is that homomorphic encryption currently only supports a very limited set of data operations, and therefore cannot yet be applied to a majority of computations.

Computations that are already parallelized and distributed across clouds can be probabilistically checked by simply replicating some or all of the sub-tasks and comparing the results for inconsistencies. Past works have therefore imbued MapReduce architectures with fault tolerance through massive replication with inconsistency resolution via majority voting (Moca et al., 2011) or distributed trust management (Khan and Hamlen, 2012b). In contrast, CloudCover focuses on certifying the significant class of computations that are not massively parallelized, including those that are inherently serial.

Remote attestation is an alternative to result checking that supplies evidence to a distrustful appraiser that an untrusted, remote target is running authorized software atop permissible hardware (Coker et al., 2011). By assuring the integrity of the remote computing environment, its results can be trusted without additional checking. Remote attestation solutions typically rely upon secure co-processors to attest hardware integrity (Trusted Computing Group, 2011; Nauman et al., 2010), and software monitors that relay cryptographically protected evidence of software integrity at load-time and in real-time as the remote computation progresses (Kil et al., 2009; Seshadri et al., 2005; Falcarin et al., 2005; M'Barka et al., 2009).

However, the evidence exhibited by remote attestation solutions is not a proof. A knowledgeable, resourceful, or lucky adversary can potentially forge false evidence to corrupt the computing environment without detection (cf., Delaune et al., 2010; Castelluccia et al., 2009;

Perrig and van Doorn, 2010). This is partly because most software monitors rely upon code obfuscation (Ceccato et al., 2013; Collberg and Nagra, 2010; Fukushima et al., 2008) or blackbox security (Hohl, 1998), which does not provide formal guarantees against reverse-engineering and corruption. An attacker with powerful reverse-engineering tools or inside knowledge of the obfuscation strategy can therefore potentially corrupt computations in ways that are not detectable by the appraiser. This invites an arms race in which attackers hone increasingly sophisticated analysis tools while defenders weave ever more complex obfuscations to bewilder them.

7.5 SilverLine

Isolation of multiusers in clouds is recognized as a significant research problem. Past work has proposed many different access control mechanisms to mutually isolate untrusted cloud jobs and their resources. At an implementation level, these can be broadly categorized into two main streams: (1) those that modify the cloud architecture or system, and (2) those that create an extra access control layer.

NetODESSA (Bellessa et al., 2011) introduces a distributed, host-level, dynamic policy monitoring system into the network layer of clouds. An administrator writes general policies for groups of nodes, from which the system infers more rules dynamically. Cloud-hosted services have also been proposed as a means to enforce end-to-end information flow control (Bacon et al., 2010). The vision involves a data tagging scheme that can enforce MAC, Information Flow Control (IFC), and RBAC policies that ensure end-to-end security for the whole data life through application-level virtualization. Airavat (Roy et al., 2010) enforces mandatory information flow control on Hadoop clouds by applying SELinux-style (National Security Agency (NSA), 2013) MAC to prevent information leaks through system resources. It additionally applies *differential privacy* to detect leaks within job input-output relations. While powerful, all of these approaches require deep modifications to the VM and/or cloud framework and implementation, which may raise barriers to adoption. In contrast, SilverLine does not modify the cloud.

DACC (Ruj et al., 2011) adopts distributed Key Distribution Centers (KDCs) and decentralized *attribute-based encryption* to provide distributed access control in clouds. CloudPolice (Popa et al., 2010) proposes an extra access control layer within the hypervisors at end-hosts. These works prevent unauthorized access to the cloud and its resources, but do not address the problem of authorized users performing operations that (intentionally or unintentionally) violate data confidentiality. Cloudtracker (Baig et al., 2013) performs side-channel detection from the VM layer to identify dangerous job behavior that malicious users could abuse to infer private information about co-located jobs. SilverLine complements these works by securing explicit information flows introduced by authorized users through non-side channels.

A long history of works mitigate application-level security breaches and intrusions by guarding the application-OS boundary, intercepting and filtering the application's access to OS-level resources (e.g., Janus (Goldberg et al., 1996), MAPbox (Acharya and Raje, 2000), and BlueBox (Chari and Cheng, 2003)). Effectively applying this sandboxing approach to cloud jobs is challenging because clouds introduce extra layers of infrastructure below the OS that have the effect of conflating permissible and impermissible operations at the OS level. For example, a job's request to write to a particular Hadoop Distributed File System (HDFS) object may only be exposed to the OS as a write to a much larger, OS-level file object that combines many HDFS objects. Monitoring at this level is therefore too coarse-grained to properly enforce many policies of interest.

SilverLine deploys IRMs (Schneider, 2000) to constrain untrusted cloud jobs. Prior research has shown that IRMs are more powerful than external execution monitors (Hamlen et al., 2006b; Ligatti et al., 2009), in part because they can observe and restrict fine-grained program behaviors that are difficult or impossible to observe by monitors implemented outside the user code. Extensive prior work has examined the problem of automatically in-lining secure policy enforcement programming into binary programs for which source code is unavailable (e.g., Erlingsson and Schneider, 1999; Hamlen et al., 2006a; Ligatti, 2006; Hamlen, 2006; Ligatti et al., 2009). One widely used technique is to express the IRM's programming as aspects in an AOP language (Kiczales et al., 1997), and apply aspect-weaving to in-line it into binary programs (Bauer et al., 2005; Chen and Roşu, 2005). This is the approach employed by SilverLine.

Subsequent research has proved that such in-lining can secure untrusted mobile code even when the code was crafted by a malicious adversary that knows all implementation details of the IRM in advance (Hamlen et al., 2006a; Aktug et al., 2008; Hamlen et al., 2012). In essence, the IRM implementation carefully leverages object encapsulation, control-flow safety, and type-safety properties of the binary language in which the mobile code is expressed, to guarantee that the surrounding untrusted code into which the IRM is in-lined cannot corrupt or circumvent the IRM's security programming at runtime.

Hamlen et al. (Hamlen et al., 2012) propose a framework to enforce security policies for cloud data management, where they discuss different possible approaches including leveraging IRMs. SilverLine is the continuation of that research initiative, and offers a full design, implementation, and evaluation in a realistic cloud environment.

CHAPTER 8
CONCLUSION

Enormous progress in hardware, networking, middleware, and virtual machine technologies has led to an emergence of new, globally distributed computing platforms that provide computation facilities and storage as services accessible from anywhere via the Internet. At the fore of this movement, cloud computing (Amazon, 2013; Microsoft, 2013; Apache, 2013) has been widely heralded as a new, promising platform for delivering information infrastructure and resources as IT services (Buyya et al., 2009; Weiss, 2007). Customers can access these services in a pay-as-you-go fashion while saving huge capital investment in their own IT infrastructure (Armbrust et al., 2009). Thus, cloud computing is now a pervasive presence of enormous importance to the future of e-commerce.

However, there are many challenges facing cloud computing to be widely deployed and used—the major one being security, which has been a vast research area in recent years. One primary concern is trust management in clouds.

Existing cloud computing platform implementations place centralized universal trust over all the cloud nodes that magnifies the concerns of data and computation integrity and security. This dissertation emphasizes decentralization of this trust relationship in clouds to overcome these major concerns for cloud users, and proposes paradigms to establish it. In this chapter, I conclude my dissertation with some future directions, for each of our established frameworks.

8.1 Hatman

Hatman decentralizes trust using replication and extends Hadoop clouds with reputation-based trust management of slave data nodes based on EigenTrust (Kamvar et al., 2003).

To obtain high scalability, all trust management computations are formulated as distributed cloud computations. This leverages the considerable computing power of the cloud to improve the data integrity of cloud computations. Experiments show that Hatman consistently obtains over 90% reliability after just 100 jobs even when 25% of the network is malicious, and scales extremely well with increased job replication rates.

Although our implementation augments a full-scale, production-level cloud system, our evaluation is preliminary. In future work we plan to extend our analysis to consider more sophisticated data integrity attacks (e.g., malicious collectives) against larger clouds. We also plan to investigate the impact of job non-determinacy on integrity attestations based on consistency-checking.

8.2 AnonymousCloud

AnonymousCloud distributes trust by dissociating ownership information from the submitted jobs to clouds and improves data privacy in the cloud by decoupling private data content from metadata concerning its provenance and semantics. Our system, AnonymousCloud, employs Tor onion routing inside cloud providers for customers to anonymously communicate computations and data to the system. An anonymous authentication system based on public-key cryptography facilitates billing of anonymous customers without linking their private data to their identities. Simulation results demonstrate that AnonymousCloud provides superior data ownership privacy even when a large percentage of the cloud is malicious.

For our future research we consider adding incentive-based congestion control to reduce the computational overhead of long Tor circuits, and cover traffic for defense against end-to-end timing attacks. In addition, greater transparency of internal cloud resources is recommended as a means of generating greater consumer confidence in cloud systems.

8.3 Penny

Penny decentralizes trust by distributing clouds master nodes trust among many peers. It efficiently supports global trust labels, data integrity labels, and data confidentiality labels in a fully decentralized, structured, peer-to-peer network. Global labeling assures convergence for all security queries, while decentralization avoids centralized points of failure typically associated with centralized label servers. Its reputation management system applies and extends EigenTrust (Kamvar et al., 2003), distributed hash tabling based on Chord (Stoica et al., 2001), and anonymizing tunnels based on Tarzan (Chaum and van Heyst, 1991; Freedman et al., 2002) or SurePath (Zhu and Hu, 2008). The security labeling scheme preserves the efficiency of network operations; lookup cost including label retrieval is $O(\log N + k)$, where N is the network size and k is a constant replication factor.

We developed a Penny client in Java and tested it under eight attack simulations. The results illustrate Penny's efficiency and reliability over realistic network operations, including high dynamic churn; object publications, lookups, and downloads; and regular reputation maintenance via the Secure EigenTrust algorithm. The results also demonstrate the robustness of Penny in the presence of malicious agents. We obtain extremely high average success rates for all experiments even when 20% of the network is malicious. Experiments show that success rates remain high even with relatively complex publish protocols, such as those used to manage RDF data.

Penny is one contribution to the larger research question of how to combine anonymity with reputation-based trust management. Anonymity and reputation-based trust are often at odds because it is difficult to divulge an agent's reputation without also divulging its identity. Penny accomplishes this by decoupling object-owner information through a cryptographically protected layer of indirection.

Our implementation and analysis did not consider attacks upon the P2P network overlay itself, such as denial of service, message misrouting, message tampering, or traffic pattern analysis. Using trust values to change the routing structure (so as to avoid routing messages through malicious agents) is an interesting and active area of research that might address these vulnerabilities (cf., Hamlen and Hamlen, 2012). We intend to consider such attacks in future work.

Future research should also consider how to enforce information flow policies based on Penny's integrity and confidentiality labeling system. For example, Penny's publish and request protocols might be augmented with security checks that block the dissemination of data items whose integrity labels lie below a certain threshold. This would have the effect of censoring known malware from the network. One might also enforce a corresponding confidentiality policy that prohibits low-trust agents from obtaining high confidentiality data, but this is a more difficult research challenge. In order to prevent future confidentiality violations the trust management system must be informed of past confidentiality violations, but it is unclear how to ensure that such violations get reported (since typically the only witnesses are the malicious agents involved in leaking the data). Enforcing strong confidentiality policies in P2P networks therefore remains an interesting open problem.

8.4 CloudCover

CloudCover is a novel approach to SECaaS that that distributes trust on the user side and allows Java computations executed in untrusted environments to be validated by commodity data processing clouds. Conceptually, it realizes proof-carrying computations as checkpoint chains. Generation and validation of such proofs is possible with relatively minor changes to existing Java software due to the insight that all Java programs already carry a notion of checkpoint equality encoded in their object-equality implementations. This serves as a computation integrity contract that can be validated by a trusted third party. The validation algorithm is massively parallelizable even when the original computation is largely serial, and can be spot-checked for even more efficient validation of probabilistic (yet quantifiable) integrity guarantees.

We demonstrate the feasibility of CloudCover's approach by implementing it and evaluating it on a real-world architecture: Hadoop MapReduce. Experimental results indicate that relatively few modifications to existing Java software are needed to add proof-carrying capabilities, and that validation services have a natural implementation as MapReduce jobs.

Our current implementation instruments Java programs with proof-carrying powers semi-manually. Future work should consider automated, binary-level approaches for doing so. In addition, our preliminary experiments consider only small-scale clouds, simple Java computations, and clients consisting of desktop machines. In the future, we intend to scale our work to larger scenarios and handheld devices, such as smart phones.

Applying our approach to other languages requires a means of generating checkpoints that can be compared for semantic equivalence. Managed, object-oriented, bytecode languages, such as Java, .NET, and ActionScript, facilitate such comparison through built-in class methods that decide object-equality. Native codes that realize checkpoints as process memory images admit such certification only if the images can be made insensitive to low-level hardware details that differ between the untrusted host and the trusted checker. Future work should investigate the feasibility of extending our work to such domains.

8.5 SilverLine

SilverLine, again decentralizing trust on user side and moving the trust to the checker, is the first cloud information flow enforcement framework whose implementation makes no alteration to the cloud infrastructure, and that is completely transparent to job authors—requiring no change to job development practices or API usage. This makes it easily implementable and adaptable to real-world clouds, since the cloud and the enforcement can be maintained completely separately and orthogonally. It achieves this by realizing the enforcement as an IRM that is in-lined into untrusted binary jobs at the cloud's edge. The resulting jobs self-monitor their accesses and collectively maintain a distributed information flow graph within the cloud, which tracks the history of flows and prohibits policy-violating operations. Well-established IRM design methodology was applied to secure the IRM against attacks from the code into which it is in-lined, protecting it even from threats that know all the IRM's implementation details.

We demonstrated the feasibility of SilverLine by implementing and evaluating it in a real cloud architecture: Hadoop MapReduce. The popular AOP language AspectJ is leveraged to elegantly formulate and instantiate IRMs within the Hadoop architecture. Experimental results illustrate the efficiency and scalability of SilverLine with low overhead.

Our present prototype is limited to enforcement of mandatory, role-based, access controls of explicit information flows between principals. Future work should examine the applicability of our approach to enforce larger, more expressive policy classes and policy languages. There are also many engineering challenges that should be investigated to optimize the approach for large-scale clouds. A prominent one is the question of how best to store and maintain global security state (e.g., the IFG) within the cloud without introducing bottlenecks for massive parallelism.

Past work has shown that the security of IRM frameworks can be strengthened by introducing a formal verification step that removes the significant complexity of the binary-rewriter from the trusted computing base (Hamlen et al., 2006a; Aktug et al., 2008; Hamlen et al., 2012). The verification step applies type-checking, contract-checking, or model-checking to the rewritten job code to automatically and independently certify that the self-monitoring job is incapable of violating the security policy when executed (i.e., the IRM precludes all possible violations). The verification algorithm's implementation is typically much smaller than the code-rewriting infrastructure (because it performs no code-generation and conservatively rejects programs whose safety is unclear), and is therefore viewed as more trustworthy. In future work we plan to investigate the feasibility of such verification for validating IRMs in the cloud.

REFERENCES

Abawajy, J. (2009). Determining service trustworthiness in intercloud computing environments. In *Proceedings of the International Symposium on Pervasive Systems, Algorithms, and Networks (I-SPAN)*, pp. 784–788.

Abbott, T., K. Lai, M. Lieberman, and E. Price (2007). Browser-based attacks on Tor. In *Proceedings of the 7th International Conference on Privacy Enhancing Technologies (PET)*, pp. 184–199.

Aberer, K. and Z. Despotovic (2001). Managing trust in a peer-2-peer information system. In *Proceedings of the 10th ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 310–317.

Acharya, A. and M. Raje (2000). MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium.*

Aktug, I., M. Dam, and D. Gurov (2008). Provably correct runtime monitoring. In *Proceedings of the 15th International Symposium on Formal Methods (FM)*, pp. 262–277.

Amazon (2013). Amazon elastic compute cloud. http://aws.amazon.com/ec2.

Anderson, D. P., J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer (2002). SETI@home: An experiment in public-resource computing. *Communications of the ACM (CACM) 45*(11), 56–61.

Apache (2013). Hadoop. http://hadoop.apache.org.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia (2009). Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, U.C. Berkeley.

Bachrach, Y., A. Parnes, A. Procaccia, and J. Rosenschein (2009). Gossip-based aggregation of trust in decentralized reputation systems. *Journal of Autonomous Agents and Multiagent Systems (AAMAS) 19*(2), 153–172.

Backes, M., J. Camenisch, and D. Sommer (2005). Anonymous yet accountable access control. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, pp. 40–46.

Bacon, J., D. Evans, D. M. Eyers, M. Migliavacca, P. Pietzuch, and B. Shand (2010). Enforcing end-to-end application security in the cloud (big ideas paper). In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, pp. 293–312.

Baig, M. B., C. Fitzsimons, S. Balasubramanian, R. Sion, and D. Porter (2013). Cloudtracker: Cloud-wide policy enforcement with real-time VM introspection. Poster at the 34th IEEE Symposium on Security & Privacy (S&P).

Barthe, G., P. Buiras, and C. Kunz (2010). A functional framework for result checking. In *Proceedings of the 9th International Symposium on Functional and Logic Programming*, pp. 72–86.

Bauer, L., J. Ligatti, and D. Walker (2005). Composing security policies with polymer. In *Proceedings of the 26th ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 305–314.

Bell, D. E. and L. J. Lapadula (1973). Secure computer systems: Mathematical foundations and model. Technical Report 2547, Vol. 1, MITRE Corporation.

Bellessa, J., E. Kroske, R. Farivar, M. Montanari, K. Larson, and R. H. Campbell (2011). NetODESSA: Dynamic policy enforcement in cloud networks. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pp. 57–61.

Berger, S., R. Caceres, D. Pendarakis, R. Sailer, E. Valdez, R. Perez, W. Schildhauer, and D. Srinivasan (2008). TVDc: Managing security in the trusted virtual datacenter. *ACM SIGOPS Operating Systems Review (OSR) 42*(1), 40–47.

Berns, A. D. and E. Jung (2008, April). Searching for malware in BitTorrent. Technical Report UICS-08-05, University of Iowa Computer Science.

Bhadauria, R. and S. Sanyal (2012). Survey on security issues in cloud computing and associated mitigation techniques. *International Journal of Computer Applications (IJCA) 47*(18), 47–66.

BitTorrent (2012). http://www.bittorrent.com.

Blaze, M., J. Feigenbaum, and J. Lacy (1996). Decentralized trust management. In *Proceedings of the 17th IEEE Symposium on Security & Privacy (S&P)*, pp. 164–173.

Blaze, M., J. Feigenbaum, and M. Strauss (1998). Compliance checking in the PolicyMaker trust management system. In *Proceedings of the 2nd International Financial Cryptography Conference (FC)*, pp. 254–274.

Blaze, M., S. Kannan, I. Lee, O. Sokolsky, J. M. Smith, A. D. Keromytis, and W. Lee (2009). Dynamic trust management. *IEEE Computer 42*(2), 44–52.

Bloch, J. (2008). *Effective Java* (2nd ed.)., Chapter 3, Item 8: Obey the general contract when overriding `equals`, pp. 33–44. Sun Microsystems.

Blum, M. and S. Kannan (1989). Designing programs that check their work. In *Proceedings of the 21st ACM Symposium on Theory of Computing (STOC)*, pp. 86–97.

Borisov, N. (2005). *Anonymous Routing in Structured Peer-to-peer Overlays*. Ph. D. thesis, University of California, Berkeley, CA.

Bowers, K. D., A. Juels, and A. Oprea (2009). Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pp. 187–198.

Buyya, R., C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. In *Future Generation Computer Systems (FGCS)*, pp. 599–616.

Cai, M., M. Frank, J. Chen, and P. Szekely (2004). MAAN: A multi-attribute addressable network for grid information services. *Journal of Grid Computing 2*, 3–14.

Cai, M., M. Frank, B. Yan, and R. MacGregor (2004). A subscribable peer-to-peer RDF repository for distributed metadata management. *Journal of Web Semantics 2*(2), 109–130.

Camenisch, J. and E. V. Herreweghen (2002). Design and implementation of the *idemix* anonymous credential system. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pp. 21–30.

Camenisch, J. and A. Lysyanskaya (2004). Signature schemes and anonymous credentials from bilinear maps. In *Proceedings of the 24th Annual International Cryptology Conference (CRYPTO)*, pp. 56–72.

Castelluccia, C., A. Francillon, D. Perito, and C. Soriente (2009). On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pp. 400–409.

Ceccato, M., M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella (2013). A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Emperical Software Engineering*, 1–35.

Chari, S. N. and P.-C. Cheng (2003). BlueBoX: A policy-driven, host-based intrusion detection system. *ACM Transactions on Information and Systems Security (TISSEC) 6*(2), 173–200.

Chaum, D. (1985). Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM (CACM) 28*(10), 1030–1044.

Chaum, D. and E. van Heyst (1991). Group signatures. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, pp. 257–265.

Chen, D. and H. Zhao (2012). Data security and privacy protection issues in cloud computing. In *Proceedings of the International Conference on Computer Science and Electronics Engineering (ICCSEE)*, pp. 647–651.

Chen, F. and G. Roşu (2005). Java-MOP: A monitoring oriented programming environment for Java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 546–550.

Chen, Y., V. Paxson, and R. H. Katz (2010, January). What's new about cloud computing security? Technical Report UCB/EECS-2010-5, Electrical Engineering and Computer Sciences, University of California at Berkeley.

Chow, R., P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina (2009). Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the ACM Workshop on Cloud Computing Security (CCSW)*, pp. 85–90.

Christodorescu, M., R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni (2009). Cloud security is not (just) virtualization security. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, pp. 97–102.

Ciaccio, G. (2006). Improving sender anonymity in a structured overlay with imprecise routing. In *Proceedings of the Privacy Enhancing Technologies Workshop (PET)*, pp. 190–207.

Cloud Security Alliance (2010, March). Top threats to cloud computing v1.0. http://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf.

Coker, G., J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen (2011). Principles of remote attestation. In *Proceedings of the 10th International Conference on Information and Communications Security (ICICS)*, Volume 10(2), pp. 63–81.

Coleman, C. (2012, September). Cloud conversion saves GSA millions. http://gsablogs.gsa.gov/gsablog/2012/09/25/cloud-conversion-saves-gsa-millions.

Collberg, C. and J. Nagra (2010). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley.

Commons, A. (2013). Javaflow. http://commons.apache.org/sandbox/javaflow.

Cooper, A. and A. Martin (2006). Towards a secure, tamper-proof grid platform. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pp. 373–380.

Cornelli, F., E. Damiani, S. di Vimercati, S. Paraboschi, and P. Samarati (2002). Choosing reputable servents in a P2P network. In *Proceedings of the 11th International World Wide Web Conference (WWW)*, pp. 376–386.

Cravedi, K. and T. Randall (2012). 1000 genomes project data available on amazon cloud. Technical report, NIH News.

Dahbur, K., B. Mohammad, and A. B. Tarakji (2011). A survey of risks, threats and vulnerabilities in cloud computing. In *Proceedings of the International Conference on Intelligent Semantic Web-services and Applications (ISWSA)*.

Damiani, E., S. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante (2002). A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pp. 207–216.

Dean, J. and S. Ghemawat (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM (CACM) 51*(1), 107–113.

Delaune, S., S. Kremer, M. D. Ryan, and G. Steel (2010). A formal analysis of authentication in the TPM. In *Proceedings of the 7th International Conference on Formal Aspects of Security and Trust (FAST)*, pp. 111–125.

Denning, D. E. (1976). A lattice model of secure information flow. *Communications of the ACM (CACM) 19*(5), 236–243.

Dingledine, R., N. Mathewson, and P. Syverson (2004). Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pp. 303–320.

Du, J., N. Shah, and X. Gu (2011). Adaptive data-driven service integrity attestation for multi-tenant cloud systems. In *Proceedings of the IEEE International Workshop on Quality of Service (IWQoS)*, pp. 1–9.

Du, J., W. Wei, X. Gu, and T. Yu (2009). Towards secure dataflow processing in open distributed systems. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, pp. 67–72.

Du, J., W. Wei, X. Gu, and T. Yu (2010). RunTest: Assuring integrity of dataflow processing in cloud computing infrastructures. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pp. 293–304.

Dwork, C. (2008). Differential privacy: A survey of results. In *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation (TAMC)*, pp. 1–19.

Erlingsson, Ú. and F. B. Schneider (1999). SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pp. 87–95.

ExpressionTech and The Tor Project (2012). Tor Cloud project. https://cloud.torproject.org/.

Falcarin, P., R. Scandariato, M. Baldi, and Y. Ofek (2005, October). Integrity checking in remote computation. In *Atti Del XLIII Congresso Annuale (AICA)*.

Freedman, M., E. Sit, J. Cates, and R. Morris (2002). Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pp. 193–206.

Fujitsu (2010). Personal data in the cloud: A global survey of consumer attitudes. Technical report, Fujitsu Research Institute.

Fukushima, K., S. Kiyomoto, and Y. Miyake (2011). Towards secure cloud computing architecture: A solution based on software protection mechanism. *Journal of Internet Services and Information Security (JISIS) 1*(1), 4–17.

Fukushima, K., S. Kiyomoto, T. Tanaka, and K. Sakurai (2008). Analysis of program obfuscation schemes with variable encoding technique. *IEICE Transactions: Special Section on Cryptography and Information Security 91-A*(1), 316–329.

Gedik, B., H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo (2008). SPADE: The System S declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1123–1134.

Gerla, M. and D. Huang (Eds.) (2012, August). *Proceedings of the 1st Edition of the MCC Workshop on Mobile Cloud Computing*. SIGCOMM Special Interest Group on Data Communication: ACM.

Gnutella (2010). http://www.gnutella.com.

Goldberg, I., D. Wagner, R. Thomas, and E. A. Brewer (1996). A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 5th USENIX Security Symposium*.

Google (2013a). Google App Engine. https://developers.google.com/appengine.

Google (2013b). Google Apps. http://www.google.com/enterprise/apps/business.

Greenberg, A. (2012, April). The Tor project's new tool aims to map out internet censorship. *Forbes*.

Guo, Y., Z. Pan, and J. Heflin (2004). An evaluation of knowledge base systems for large OWL datasets. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, pp. 274–288.

Guo, Y., Z. Pan, and J. Heflin (2005). LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics 3*(2–3), 158–182.

Gupta, M., P. Judge, and M. Ammar (2003). A reputation system for peer-to-peer networks. In *Proceedings of the 13th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pp. 144–152.

Hamlen, K. and W. Hamlen (2012). An economic perspective of message-dropping attacks in peer-to-peer overlays. *Intelligence and Security Informatics (ISI) 1*(6).

Hamlen, K. W. (2006, August). *Security Policy Enforcement by Automated Program-rewriting.* Ph. D. thesis, Cornell University.

Hamlen, K. W. and M. Jones (2008). Aspect-oriented in-lined reference monitors. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pp. 11–20.

Hamlen, K. W., M. M. Jones, and M. Sridhar (2012). Aspect-oriented runtime monitor certification. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 126–140.

Hamlen, K. W., L. Kagal, and M. Kantarcioglu (2012). Policy enforcement framework for cloud data management. *IEEE Data Engineering Bulletin (DEB), Special Issue on Security and Privacy in Cloud Computing 35*(4), 39–45.

Hamlen, K. W., M. Kantarcioglu, L. Khan, and B. Thuraisingham (2010). Security issues for cloud computing. *International Journal of Information Security and Privacy (IJISP) 4*(2), 36–48.

Hamlen, K. W., G. Morrisett, and F. B. Schneider (2006a). Certified in-lined reference monitoring on .NET. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pp. 7–16.

Hamlen, K. W., G. Morrisett, and F. B. Schneider (2006b). Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages And Systems (TOPLAS) 28*(1), 175–205.

He, Q., J. Yan, H. Jin, and Y. Yang (2009). ServiceTrust: Supporting reputation-oriented service selection. In *Proceedings of the 7th International Joint Conference on Service-oriented Computing*, pp. 269–284.

Hohl, F. (1998). Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pp. 92–113.

Holden, W. (2010, January). Mobile cloud applications & services: Monetising enterprise & consumer markets 2009–2014. Technical report, Juniper Research.

Hopper, N., E. Y. Vasserman, and E. Chan-Tin (2010). How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC) 13*(2).

Huang, R., X. Gui, S. Yu, and W. Zhuang (2011). Research on privacy-preserving cloud storage framework supporting ciphertext retrieval. In *Proceedings of the International Conference on Network Computing and Information Security (NCIS)*, pp. 93–97.

Iannotta, B. (2011, November). Securing the cloud: The intel community's high-stakes bid to keep its data safe. *C4ISR Journal, DefenseNews*.

Ibrahim, A. S., J. Hamlyn-Harris, J. Grundy, and M. Almorsy (2011). CloudSec: A security monitoring appliance for virtual machines in the IaaS cloud model. In *Proceedings of the 5th International Conference on Network and System Security (NSS)*, pp. 113–120.

Jakubowski, M., R. Venkatesan, and Y. Yacobi (2009). Quantifying trust. Technical Report MSR-TR-2009-119, Microsoft Corporation.

Jensen, M., S. Schäge, and J. Schwenk (2010). Towards an anonymous access control and accountability scheme for cloud computing. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pp. 540–541.

Kamvar, S., M. Schlosser, and H. Garcia-Molina (2003). The EigenTrust algorithm for reputation management in P2P networks. In *Proceedings of the 12th International World Wide Web Conference (WWW)*, pp. 640–651.

KaZaA (2012). http://www.kazaa.com.

Khaled, A., M. F. Husain, L. Khan, K. W. Hamlen, and B. Thuraisingham (2010). A token-based access control system for RDF data in the clouds. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 104–111.

Khan, S. M. and K. W. Hamlen (2012a). AnonymousCloud: A data ownership privacy provider framework in cloud computing. In *Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 170–176.

Khan, S. M. and K. W. Hamlen (2012b). Hatman: Intra-cloud trust management for Hadoop. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 494–501.

Khan, S. M. and K. W. Hamlen (2013a). Computation certification as a service in the cloud. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).*

Khan, S. M. and K. W. Hamlen (2013b). Penny: Secure, decentralized data management. *International Journal of Network Security (IJNS) 16*(4), 289–303.

Khan, S. M., K. W. Hamlen, and M. Kantarcioglu (2014). Silver lining: Enforcing secure information flow at the cloud edge. Submitted for publication.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G.Griswold (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pp. 327–353.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pp. 220–242.

Kil, C., E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang (2009). Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 115–124.

Ko, R., P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. Lee (2011). TrustCloud: A framework for accountability and trust in cloud computing. In *Proceedings of the IEEE World Congress on Services (SERVICES)*, pp. 584–588.

Kotla, R., L. Alvisi, M. Dahlin, A. Clement, and E. Wong (2009). Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions On Computer Systems (TOCS) 27*(4).

Kulkarni, G., N. Chavan, R. Chandorkar, R. Waghmare, and R. Palwe (2012). Cloud security challenges. In *Proceedings of the 7th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*, pp. 88–91.

Laurikainen, R. (2010). Secure and anonymous communication in the cloud. Technical Report TKK-CSE-B10, Aalto University School of Science and Technology, Department of Computer Science and Engineering.

Lee, S., R. Sherwood, and B. Bhattacharjee (2003). Cooperative peer groups in NICE. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer Communications (INFOCOM)*, pp. 1272–1282.

Li, W. and L. Ping (2009). Trust model to enhance security and interoperability of cloud environment. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*, pp. 69–79.

Liarou, E., S. Idreos, and M. Koubarakis (2007a). Continuous RDF query processing over DHTs. In *Proceedings of the 6th International the Semantic Web and 2nd Asian Semantic Web Conference (ISWC/ASWC)*, pp. 324–339.

Liarou, E., S. Idreos, and M. Koubarakis (2007b). Publish/Subscribe with RDF data over large structured overlay networks. In *Proceedings of the 2005/2006 International Conference on Databases, Information Systems, and Peer-to-peer Computing (DBISP2P)*, pp. 135–146.

Ligatti, J., L. Bauer, and D. Walker (2009). Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC) 12*(3).

Ligatti, J. A. (2006, June). *Policy Enforcement Via Program Monitoring*. Ph. D. thesis, Princeton University.

Lindell, Y. and B. Pinkas (2009). Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality 1*(1), 59–98.

Liu, Q., G. Wang, and J. Wu (2012). Secure and privacy preserving keyword searching for cloud storage services. *Journal of Network and Computer Applications (JNCA) 35*(3), 927–933.

Manuel, P. D., S. Thamarai Selvi, and M.-E. Barr (2009). Trust management system for grid and cloud resources. In *Proceedings of the International Conference on Advanced Computing (ADCONS)*, pp. 176–181.

Marozzo, F., D. Talia, and P. Trunfio (2010). A peer-to-peer framework for supporting MapReduce applications in dynamic cloud environments. In N. Antonopoulos and L. Gillam (Eds.), *Cloud Computing: Principles, Systems and Applications*, pp. 113–125. Springer.

Marsh, S. (1994, April). *Formalising Trust as a Computational Concept*. Ph. D. thesis, University of Stirling, Scotland, UK.

M'Barka, M. B., F. Krief, and O. Ly (2009). Entrusting remote software executed in an untrusted computation helper. In *Proceedings of the International Conference on Network and Service Security (N2S)*, pp. 1–5.

Meijer, G. (2012). 5 cloud computing statistics. Technical report, Infograph.

Microsoft (2013). Windows Azure. http://www.windowsazure.com.

Mitchell, C. (Ed.) (2005). *Trusted Computing*. The Institution of Engineering and Technology.

Moca, M., G. C. Silaghi, and G. Fedak (2011). Distributed results checking for MapReduce in volunteer computing. In *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPSW)*, pp. 1847–1854.

Molina-Estolano, E., C. Maltzahn, B. Reed, and S. A. Brandt (2010). Haceph: Scalable metadata management for Hadoop using Ceph. Poster at the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI).

Mondal, J. and A. Deshpande (2012). Managing large dynamic graphs efficiently. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 145–156.

Morsy, M. A., J. Grundy, and I. Müller (2010). An analysis of the cloud computing security problem. In *Proceedings of the 1st Asia-Pacific Workshop on Cloud Computing (APSEC-CLOUD).*

Nakajima, J. and M. Matsui (2002). Performance analysis and parallel implementation of dedicated hash functions. In *Proceedings of the Annual International Conference on Theory and Application of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT)*, pp. 165–180.

Napster (2012). http://www.napster.com.

National Institute of Standards and Technology (1995, April). Secure hash standard. Federal Information Processing Standard FIPS-180-1.

National Security Agency (NSA) (2013). SELinux. http://selinuxproject.org.

Nauman, M., S. Khan, X. Zhang, and J.-P. Seifert (2010). Beyond kernel-level integrity measurement: Enabling remote attestation for the Android platform. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST)*, pp. 1–15.

Necula, G. C. and P. Lee (1998). Safe, untrusted agents using proof-carrying code. In *Proceedings of Mobile Agents and Security*, pp. 61–91.

Nepal, S., C. Shiping, Y. Jinhui, and D. Thilakanathan (2011). DIaaS: Data integrity as a service in the cloud. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pp. 308–315.

Newman, A., J. Hunter, Y. Li, C. Bouton, and M. Davis (2008). A scale-out RDF molecule store for distributed processing of biomedical data. In *Proceedings of the Semantic Web for Health Care and Life Sciences Workshop (HCLS) at the 17th International World Wide Web Conference (WWW).*

Newman, A., Y. Li, and J. Hunter (2008). A scale-out RDF molecule store for improved co-identification, querying and inferencing. In *Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS) at the 7th International Semantic Web Conference (ISWC).*

Nguyen, T. and S. Weisong (2010). Improving resource efficiency in data centers using reputation-based resource selection. In *Proceedings of the International Conference on Green Computing (ICGREEN)*, pp. 389–396.

Nicholson, B., A. Owrak, and L. Daly (2013). Cloud computing research. Technical report, Manchester Business School, Commissioned By Rackspace.

Pearson, S. and A. Benameur (2010). Privacy, security and trust issues arising from cloud computing. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 693–702.

Pearson, S., Y. Shen, and M. Mowbray (2009). A privacy manager for cloud computing. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pp. 90–106.

Perrig, A. and L. van Doorn (2010). Refutation of 'On the difficulty of software-based attestation of embedded devices'. http://sparrow.ece.cmu.edu/group/pub/perrig-vandoorn-refutation.pdf.

Pfitzmann, A. and M. Hansen (2010, August). A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. http://dud.inf.tu-dresden.de/Anon_Terminology.shtml. v0.34.

Ponemon Institute (2013, June). The risk of regulated data on mobile devices & in the cloud. Technical report, Ponemon Institute.

Popa, L., M. Yu, S. Y. Ko, S. Ratnasamy, and I. Stoica (2010). CloudPolice: Taking access control out of the network. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets).*

Ratnasamy, S., P. Francis, M. Handley, R. Karp, and S. Schenker (2001). A scalable, content-addressable network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer and Communications (SIGCOMM)*, pp. 161–172.

Rivest, R., A. Shamir, and Y. Tauman (2001). How to leak a secret. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pp. 552–565.

Rivest, R. L. (1992, April). The MD5 message digest algorithm. Internet RFC 1321.

Rowstron, A. and P. Druschel (2001). Pastry: Scalable, decentralized object location and routing for large scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pp. 329–350.

Roy, I., S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel (2010). Airavat: Security and privacy for MapReduce. In *Proceedings of the 7th USENIX Symposium on Networked System Design and Implementation (NSDI)*, pp. 297–312.

Ruj, S., A. Nayak, and I. Stojmenovic (2011). DACC: Distributed access control in clouds. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pp. 91–98.

Ryan, M. D. (2011). Cloud computing privacy concerns on our doorstep. *Communications of the ACM (CACM) 54*(1), 36–38.

Sabater, J. and C. Sierra (2002). Reputation and social network analysis in multi-agent systems. In *Proceedings of the 1st ACM International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 475–482.

Sabelfeld, A. and A. C. Myers (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21*(1), 5–19.

SalesForce.com (2013). Sales Force. http://www.salesforce.com.

Samarati, P. (2001). Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 13*(6), 1010–1027.

Sander, T. and C. F. Tschudin (1997). Protecting mobile agents against malicious hosts. In G. Vigna (Ed.), *Proceedings of Mobile Agents and Security*, pp. 44–60.

Sander, T. and C. F. Tschudin (1998). Towards mobile cryptography. In *Proceedings of the 19th IEEE Symposium on Security & Privacy (S&P)*, pp. 215–224.

Sandhu, R. S. (1993). Lattice-based access control models. *IEEE Computer 26*(11), 9–19.

Santos, N., K. P. Gummadi, and R. Rodrigues (2009). Towards trusted cloud computing. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud).*

Schneider, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and Systems Security (TISSEC) 3*(1), 30–50.

Schulze, H. and K. Mochalski (2009). Internet study 2008/2009. Technical report, Ipoque. http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf.

Seshadri, A., M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla (2005). Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pp. 1–16.

Shvachko, K. V. (2010). HDFS scalability: The limits of growth. *;login: 52*(2), 6–16.

Slamanig, D. (2011). More privacy for cloud users: Privacy-preserving resource usage in the cloud. In *Selected Papers from the 4th Hot Topics in Privacy Enhancing Technologies (HotPETs)*, pp. 15–27.

Smith, D. M., Y. V. Natis, G. Petri, T. J. Bittman, E. Knipp, P. Malinverno, and J. Feiman (2011, December). Predicts 2012: Cloud computing is becoming a reality. Technical Report G00226103, Gartner.

Soanes, C. and A. Stevenson (2006). *Concise Oxford English Dictionary.* Oxford University Press.

Stoica, I., R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer and Communications (SIGCOMM)*, pp. 149–160.

Subashini, S. and V. Kavitha (2011). A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications (JNCA) 34*(1), 1–11.

Sullivan, G. F., D. S. Wilson, and G. M. Masson (1995). Certification of computational results. *IEEE Transactions on Computers (TC) 44*(7), 833–847.

Swamy, N., B. J. Corcoran, and M. Hicks (2008). Fable: A language for enforcing user-defined security policies. In *Proceedings of the 29th IEEE Symposium on Security & Privacy (S&P)*, pp. 369–383.

Syverson, P. (1994). A taxonomy of replay attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop (CSFW)*, pp. 187–191.

Takabi, H., J. Joshi, and G.-J. Ahn (2010). Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy 8*(6), 24–31.

Talwalkar, A. S. (2011, January). HadoopT – breaking the scalability limits of Hadoop. Master's thesis, Rochester Institute of Technology.

Trusted Computing Group (2011, August). TCG attestation PTS protocol: Binding to TNC IF-M, version 1.0, revision 28.

Tsybulnik, N., K. Hamlen, and B. Thuraisingham (2007). Centralized security labels in decentralized P2P networks. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pp. 315–324.

U.S. Department of Health & Human Services (2013). Health information privacy. http://www.hhs.gov/ocr/privacy/index.html.

Vaquero, L. M., L. Rodero-Merino, and D. Morán (2011). Locking the sky: A survey on IaaS cloud security. *Computing 91*(1), 93–118.

Vasudevan, A., E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune (2012). Trustworthy execution on mobile devices: What security properties can my mobile platform give me? In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, pp. 159–178.

Verheijen, W. (2008). Efficient query processing in distributed RDF databases. Master's thesis, Technische Universiteit Eindhoven, Netherlands.

VMWare (2013). Cloud foundry. http://en.wikipedia.org/wiki/Cloud_Foundry.

Wang, F., Y. Zhang, and J. Ma (2010). Modelling and analyzing passive worms over unstructured peer-to-peer networks. *International Journal of Network Security (IJNS) 11*(1), 39–45.

Weiss, A. (2007). Computing in the cloud. In *ACM Networker*, pp. 16–25.

West, A. G., S. Kannan, I. Lee, and O. Sokolsky (2010). An evaluation framework for reputation management systems. In Z. Yan (Ed.), *Trust Modeling and Management in Digital Environments: From Social Concept to System Development*, pp. 282–308. IGI Global.

Wilcox, J. (2010). Gartner: Most CIOs have their heads in the clouds. http://betanews.com/2011/01/24/gartner-most-cios-have-their-heads-in-the-clouds.

Xiao, Z. and Y. Xiao (2012). Security and privacy in cloud computing. *IEEE Communications Surveys & Tutorials 15*(2), 843–859.

Xiong, L. and L. Liu (2004). PeerTrust: Supporting reputation-based trust in peer-to-peer communities. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 16*(7), 843–857.

Xu, J.-S., R.-C. Huang, W.-M. Huang, and G. Yang (2009). Secure document service for cloud computing. In *Proceedings of the 1st International Conference on Cloud Computing (CloudCom)*, pp. 541–546.

Zacharia, G. and P. Maes (2000). Trust management through reputation mechanisms. *Applied Artificial Intelligence 14*(9), 881–907.

Zarandioon, S., D. Yao, and V. Ganapathy (2011). K2C: Cryptographic cloud storage with lazy revocation and anonymous access. In *Proceedings of the 7th International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, pp. 491–510.

Zhang, Y., Z. Zheng, and M. R. Lyu (2011). BFTCloud: A byzantine fault tolerance framework for voluntary-resource cloud computing. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pp. 444–451.

Zhao, B., L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications (JSAC) 22*(1), 41–53.

Zhu, Y. and Y. Hu (2008). SurePath: An approach to resilient anonymous routing. *International Journal of Network Security (IJNS) 6*(2), 201–210.

VITA

Safwan Mahmud Khan earned his B.Sc. degree in Computer Science and Engineering from the University of Dhaka in Bangladesh in 2005. Following that, Safwan commenced his academic quest in the United States by obtaining an M.S. in 2008 in Computer Science from The University of Texas at Arlington where he worked in the Information Security (iSec) Lab with his advisor, Dr. Matthew Wright, in the research area related to anonymity in large scale peer-to-peer systems. Thereafter he entered the Ph.D. program in the Fall of 2009 at The University of Texas at Dallas in the Software Security Lab working under the supervision of Dr. Kevin Hamlen. Safwan's main research area has been cloud computing security and its extension to secure peer-to-peer systems. During his extensive research work in his academic career he has published seven international conference papers (plus one more under submission) and two journal papers. His research interests include security in cloud computing, secure peer-to-peer systems, network security and privacy, data mining, and algorithms. He showed his strong programming fascination by achieving 4th position in ACM ICPC (international programming contest), 2002, Dhaka, Bangladesh site.

Safwan has more than three years of solid industry experience. He started his industry career with GrameenPhone (local brand of Telenor, Norway)—the leading multinational telecommunication service provider in Bangladesh. During his higher studies in USA, Safwan was involved with Lever-it, a software company in Mexico - for whom he worked remotely from Dallas for their Mexican office, followed by two intense internship experiences with BlackBerry and eBay.