Silver Lining: Enforcing Secure Information Flow at the Cloud Edge

Safwan Mahmud Khan, Kevin W. Hamlen and Murat Kantarcioglu
Department of Computer Science
University of Texas at Dallas
Richardson, Texas, USA
{*safwan,hamlen,muratk*}@*utdallas.edu*

***Abstract*—SilverLine is a novel, exceptionally modular framework for enforcing mandatory information flow policies for Java computations on commodity, data-processing, Platform-as-a-Service 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, VM, or cloud file system, SilverLine automatically in-lines secure information flow tracking code into untrusted Java 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 (i.e., modularly). To demonstrate the approach's feasibility, a prototype implements and deploys SilverLine on a real-world data processing cloud—Hadoop MapReduce. Evaluation results demonstrate that SilverLine provides inter-process information flow security for Hadoop clouds with easy maintainability (through modularity) and low overhead.**

***Keywords*-Access control; Aspect-Oriented Programming; Cloud computing; Information flow control; In-lined Reference Monitors; Security**

I. Introduction

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., [1]–[4]) 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 a cost savings of almost $2 million USD per year since migrating from Lotus Notes to Google's cloud-based email [5]. 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 [6].

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 [7]. 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., [8]–[10]), protecting cloud customer privacy (e.g., [11]–[13]), secure data storage in the cloud (e.g., [14], [15]), and detection and prevention of software tampering (e.g., [16], [17]).

A common challenge faced by many of these efforts is the *multitenant* problem [18]. 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 [19]. 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., [20]). 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., [21], [22]) or the underlying OS (e.g., [23]). Others add an extra access control layer atop an existing architecture, requiring new protocols (e.g., [24], [25]). 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 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 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 Mandatory Access Control (MAC) and information flow security policies on untrusted Java jobs binaries for Hadoop clouds [4], 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 enforcement as an *In-lined Reference Monitor* (IRM) [26] 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 MAC policies that restrict explicit information flows between cloud users, 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.

While Hadoop can already enforce standard information isolation policies via traditional system-level file access controls, it could not easily enforce information flow controls for multiple, mutually-distrusting tenants prior to our work. Our work therefore opens new application areas for Hadoop to which Hadoop was unsuited previously. In general, we consider the IRM approach to be well suited for enforcing many important data confidentiality and integrity policies [26]–[35] for which Hadoop and similar clouds do not yet enjoy widespread support.

SilverLine leverages *Aspect-Oriented Programming* (AOP) [36] 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 Hadoop 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. IRMs are implemented as AspectJ [37] *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.

The rest of the paper is organized as follows. Related work is summarized in §II. Section III presents SilverLine system details along with our threat model. We discuss implementation, and report results with analysis in §IV and §V, respectively. Section VI concludes with a summary of outcomes and future directions.

II. Related Work

Isolation of tenants 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 [21] 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 [22]. The vision applies data tagging to enforce MAC, Information Flow Control (IFC), and Role-based Access Control (RBAC) policies that ensure end-to-end security for the whole data life through application-level virtualization. Airavat [23] enforces mandatory information flow control on Hadoop clouds by applying SELinux-style [38] 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 [24] adopts distributed Key Distribution Centers (KDCs) and decentralized *attribute-based encryption* to provide distributed access control in clouds. CloudPolice [25] 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 [39] 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 [40], MAPbox [41], and BlueBox [42]). 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 [26] to constrain untrusted Hadoop cloud jobs. In general, IRMs are strictly more powerful than external execution monitors, 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 [28], [34]. Extensive prior work has examined the problem of automatically in-lining secure policy enforcement programming into legacy source codes of server applications [43] and general source codes expressed in security-typed languages [44], as well as into legacy binary programs for which source code is unavailable (e.g., [29]–[31], [34], [45]). One widely used technique for the latter domain expresses the IRM's programming as aspects in an AOP language [36], and applies aspect-weaving to efficiently in-line it into untrusted binary programs [27], [46]. This is the approach employed by SilverLine.

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 [29], [33], [35]. 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. [47] 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.

III. System Overview

A. 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 [48] and Google Apps [49].
- *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 [50] and Google App Engine [3].
- *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 [1] and Windows Azure [2].

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., [1], [2], [4]), 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 may violate users' information flow requirements, motivating a strong, flexible approach to enforcing such policies.

We deploy our system on Hadoop MapReduce [4], 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 user file accesses. It also dispatches and monitors the computation throughout 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.

B. 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, 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 [38] 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 [23]. SilverLine does so without modifying the cloud.

C. 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., [51]), 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. (This limitation could be lifted by employing a system like Robusta [52] to secure the native code interface.)

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.

```
maywrite user1 file2;        mayread user5 file13;
mayread user1 file4;         maywrite user5 file15;
maywrite user2 file6;        maywrite user5 file17;
maywrite user2 file9;        noflow file4 file2;
mayread user2 file12;        noflow user1 file14;
mayread user3 file1;         noflow file12 user1;
mayread user3 file3;         noflow user1 file15;
maywrite user3 file7;        noflow file8 user3;
mayread user4 file8;         noflow file18 user4;
maywrite user4 file10;       noflow file20 file16;
mayread user5 file11;        noflow file5 file19;
```

Figure 1. Sample Policy File

D. MapReduce Paradigm

MapReduce [53] 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. [1], [4]). 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.

E. Policy Specifications and Flow Tracking

An administrator-specified *policy file*, such as the one in Fig. 1, defines the specific access control and information flow policy enforced by SilverLine. Rule "maywrite P_1 P_2" grants principal P_1 direct write-access to P_2, and "mayread P_2 P_1" grants P_2 direct read-access to P_1. (At a policy level, these two commands have identical semantics; they both pemit direct information flows from P_1 to P_2.) Rule "noflow P_1 P_2" disallows information flows from P_1 to P_2. For example, in Fig. 1, even though user1 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 Fig. 1 is simplistic but suffices for our experiments. It can be conceptualized as a classic Bell-LaPadula labeling system [54] where the labels form a lattice of principal subsets (expressing impermissible flow destinations) ordered by subset relation [55], [56]. Much more expressive policy languages (e.g., [32], [57]) are possible, but supporting them is a subject of future work.

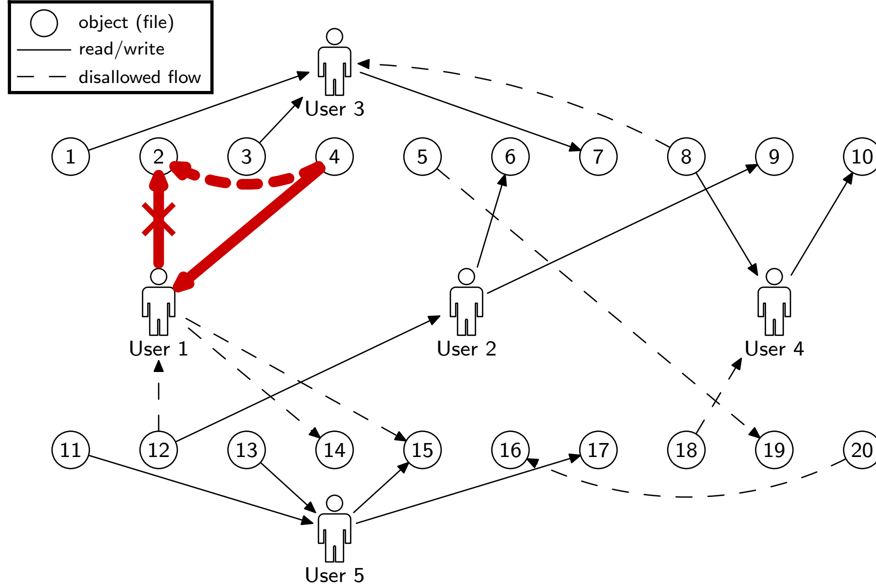

Figure 2. A sample Information Flow Graph (IFG). Edge $(\mathsf{user1}, \mathsf{file2})$ is rejected because its addition would complete disallowed flow $(\mathsf{file4}, \mathsf{file2})$.

To enforce the policy, SilverLine maintains an *Information Flow Graph (IFG)* stored as a distributed data structure in HDFS. Figure 2 depicts an example IFG. Nodes are labeled with principals (*viz.*, 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 Fig. 1, node user1 is labeled with must-not set $MN[\mathsf{user1}] = \{\mathsf{file14}, \mathsf{file15}\}$. 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 Fig. 2 show such a policy violation. The addition of edge $(\mathsf{user1}, \mathsf{file2})$ to the graph would complete disallowed flow $(\mathsf{file4}, \mathsf{file2})$, so user1'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.

F. In-lined Reference Monitor Design

SilverLine's main implementation is an AspectJ [37] 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 [28].) 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 may try to erase or overwrite the IRM code at runtime.
2) Malicious jobs may try to corrupt the IRM's internal variables or data.
3) Malicious jobs may 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

Algorithm 1 Initialize IFG

```
1: for (noallow x y) ∈ policy do
2:     MN[x] ← MN[x] ∪ {y}
3: end for
```

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 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 [29], [33], [35].

G. Architecture and Protocol

Figure 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 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 and jobs.

The IFG initialization and IRM runtime guard code are summarized in Algorithms 1 and 2, respectively. Line 16 of Algorithm 2 computes the IFG subset reachable from y, which is typically small due to the IFG's disconnectedness (see §III-E). Line 20 expresses IFGs as multisets, since duplicate edges can arise from multiply opened file handles.

Algorithm 2 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 ← o
6:         y ← job_identifier
7:     else // op open/closes o for writing
8:         x ← job_identifier
9:         y ← o
10:    end if
11:    if op is a close then
12:        IFG ← IFG − {(x, y)}
13:    else if (maywrite x y), (mayread y x) ∉ policy then
14:        throw SecurityException
15:    else if (x, y) ∉ IFG then
16:        R ← breadth_first_search(IFG, y) // R is small
17:        if R ∩ MN[x] ≠ ∅ then
18:            throw SecurityException
19:        else
20:            IFG ← IFG ⊎ {(x, y)}
21:            for z ∈ R do
22:                MN[z] ← MN[z] ∪ MN[x]
23:            end for
24:        end if
25:    end if
26: end if
```

IV. 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. Experiments were conducted on a Hadoop cluster consisting of 12 DataNodes and 1 NameNode. Nodes have Intel Pentium IV 2.40–3.00GHz processors with 2–4GB of memory each, running Ubuntu operating systems.

As mentioned in §III-F, SilverLine implements IRMs using AspectJ [37]. The total implementation size is approximately 1.5K lines of source code. Listing 1 shows a (simplified) sample aspect that implements part of Algorithm 2.

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 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.

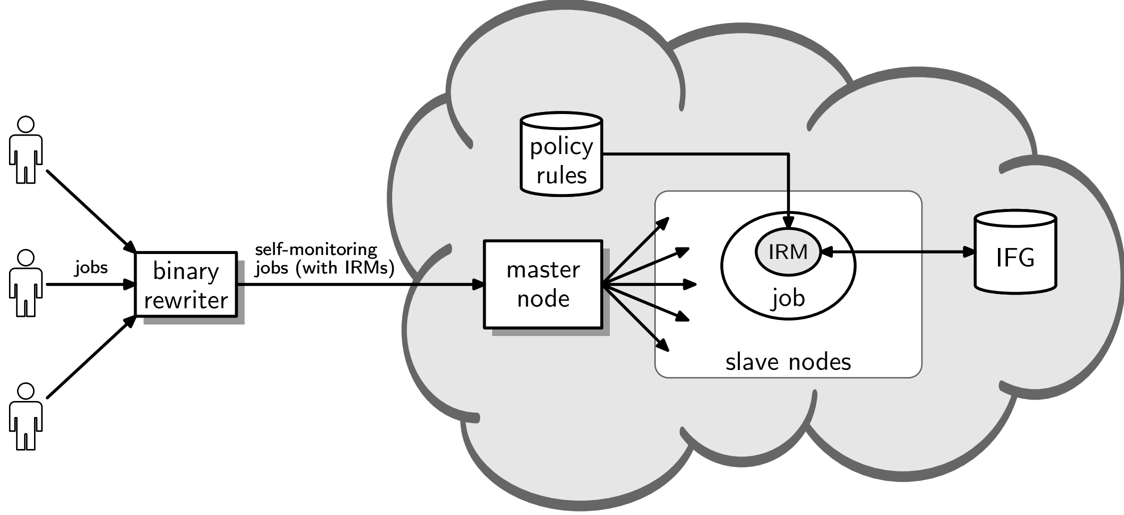

Figure 3. System architecture of SilverLine

```
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 1. Sample aspect in AspectJ

To maintain persistent history and track concurrent jobs (§III-G), 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 secrets known by U at the time U submitted J. Whenever J opens a file F_1 for reading, we join (union) [56] $MN[F_1]$ 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 graph representations are possible (cf. [58]), but are left as future work.

V. 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 principals and include attacks (§III-C) 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 violations. 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 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 4.6–7.5% slower (approximately 1.56–2.64 seconds slower) each. This slowdown is primarily due to

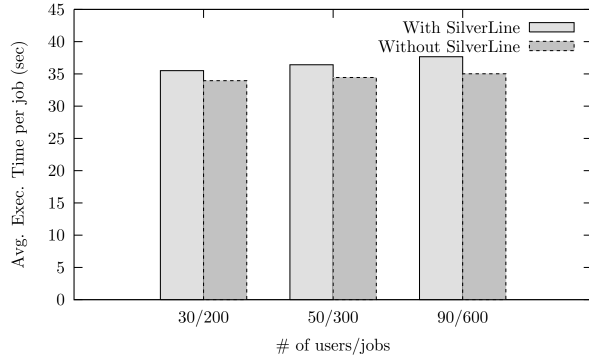

Figure 4. Performance measurement with increasing concurrency

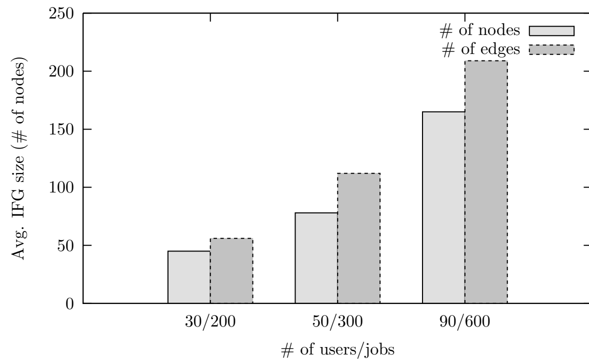

Figure 5. Performance measurement with increasing IFG size

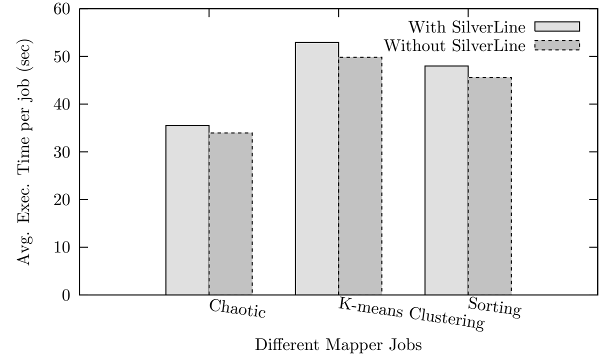

Figure 6. Comparing performance of different MapReduce jobs

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 1), the overhead remains reasonable.

Figure 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 (Fig. 6) examines performance under more practical conditions that involve some common, pre-existing MapReduce programs used in the field. We 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 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. Aspect-weaving also yielded negligible size increases to job binaries. (The size increases were masked by padding in the binary format, so are too small to be measurable.)

VI. Conclusion

SilverLine is the first Hadoop cloud information flow enforcement framework whose implementation makes no alteration to the cloud infrastructure, and that is completely transparent to Java job authors—requiring no change to the job computation language (Java bytecode) or its API. This makes it easily implementable and adaptable to real-world clouds, since the cloud and the enforcement can be maintained completely 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. Future work should consider extending the approach to workflow computations expressed in other languages, such as native code. Native code IRMs [59], [60] are a natural foundation for such extensions.

Our present prototype is limited to enforcement of mandatory 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 [29], [33], [35]. 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.

ACKNOWLEDGMENTS

The authors thank our shepherd, David Eyers, as well as the anonymous reviewers for their expert feedback, which greatly improved the paper. The research reported herein was supported in part by NSF grant CNS-1228198 and AFOSR grant FA9550-12-1-0044.

REFERENCES

[1] Amazon, "Amazon elastic compute cloud (Amazon EC2)," http://aws.amazon.com/ec2, 2013.

[2] Microsoft, "Windows Azure: Cloud computing," http://www.windowsazure.com, 2013.

[3] Google, "Google App Engine," https://developers.google.com/appengine, 2013.

[4] Apache, "Apache Hadoop," http://hadoop.apache.org, 2013.

[5] C. Coleman, "Cloud conversion saves GSA millions," http://gsablogs.gsa.gov/gsablog/2012/09/25/cloud-conversion-saves-gsa-millions, Sep. 2012.

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

[7] D. M. Smith, Y. V. Natis, G. Petri, T. J. Bittman, E. Knipp, P. Malinverno, and J. Feiman, "Predicts 2012: Cloud computing is becoming a reality," Gartner, Tech. Rep. G00226103, Dec. 2011.

[8] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling data in the cloud: Outsourcing computation without outsourcing control," in *Proc. ACM Workshop Cloud Computing Security*, 2009, pp. 85–90.

[9] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proc. Conf. Hot Topics in Cloud Computing (HotCloud)*, 2009.

[10] S. M. Khan and K. W. Hamlen, "Hatman: Intra-cloud trust management for Hadoop," in *Proc. IEEE Int. Conf. Cloud Computing (CLOUD)*, 2012, pp. 494–501.

[11] H. Takabi, J. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.

[12] S. Pearson, "Taking account of privacy when designing cloud computing services," in *Proc. ICSE Workshop Software Engineering Challenges of Cloud Computing*, 2009, pp. 44–52.

[13] S. M. Khan and K. W. Hamlen, "AnonymousCloud: A data ownership privacy provider framework in cloud computing," in *Proc. IEEE Int. Conf. Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2012, pp. 170–176.

[14] S. Nepal, C. Shiping, Y. Jinhui, and D. Thilakanathan, "DIaaS: Data integrity as a service in the cloud," in *Proc. IEEE Int. Conf. Cloud Computing (CLOUD)*, 2011, pp. 308–315.

[15] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proc. ACM Conf. Computer and Communications Security (CCS)*, 2009, pp. 187–198.

[16] K. Fukushima, S. Kiyomoto, and Y. Miyake, "Towards secure cloud computing architecture: A solution based on software protection mechanism," *J. Internet Services and Information Security (JISIS)*, vol. 1, no. 1, pp. 4–17, 2011.

[17] S. M. Khan and K. W. Hamlen, "Computation certification as a service in the cloud," in *Proc. IEEE/ACM Int. Sym. Cluster, Cloud and Grid Computing (CCGrid)*, 2013, pp. 434–441.

[18] L. M. Vaquero, L. Rodero-Merino, and D. Morán, "Locking the sky: A survey on IaaS cloud security," *Computing*, vol. 91, no. 1, pp. 93–118, 2011.

[19] Ponemon Institute, "The risk of regulated data on mobile devices & in the cloud," Ponemon Institute, Tech. Rep., Jun. 2013.

[20] U.S. Department of Health & Human Services, "Health information privacy," http://www.hhs.gov/ocr/privacy/index.html, 2013.

[21] J. Bellessa, E. Kroske, R. Farivar, M. Montanari, K. Larson, and R. H. Campbell, "NetODESSA: Dynamic policy enforcement in cloud networks," in *Proc. IEEE Sym. Reliable Distributed Systems Workshops (SRDSW)*, 2011, pp. 57–61.

[22] J. Bacon, D. Evans, D. M. Eyers, M. Migliavacca, P. Pietzuch, and B. Shand, "Enforcing end-to-end application security in the cloud (big ideas paper)," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2010, pp. 293–312.

[23] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for MapReduce," in *Proc. USENIX Conf. Networked Systems Design and Implementation (NSDI)*, 2010.

[24] S. Ruj, A. Nayak, and I. Stojmenovic, “DACC: Distributed access control in clouds,” in *Proc. IEEE Sym. Security & Privacy (S&P)*, 2011, pp. 91–98.

[25] L. Popa, M. Yu, S. Y. Ko, S. Ratnasamy, and I. Stoica, “CloudPolice: Taking access control out of the network,” in *Proc. ACM SIGCOMM Workshop Hot Topics in Networks (Hotnets)*, 2010.

[26] F. B. Schneider, “Enforceable security policies,” *ACM Trans. Information and Systems Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[27] L. Bauer, J. Ligatti, and D. Walker, “Composing security policies with polymer,” in *Proc. ACM Conf. Programming Language Design and Implementation (PLDI)*, 2005, pp. 305–314.

[28] K. W. Hamlen, G. Morrisett, and F. B. Schneider, “Computability classes for enforcement mechanisms,” *ACM Trans. Programming Languages And Systems (TOPLAS)*, vol. 28, no. 1, pp. 175–205, 2006.

[29] ——, “Certified in-lined reference monitoring on .NET,” in *Proc. ACM SIGPLAN Workshop Programming Languages and Analysis for Security (PLAS)*, 2006, pp. 7–16.

[30] J. A. Ligatti, “Policy enforcement via program monitoring,” Ph.D. dissertation, Princeton University, Jun. 2006.

[31] K. W. Hamlen, “Security policy enforcement by automated program-rewriting,” Ph.D. dissertation, Cornell University, Ithaca, New York, Aug. 2006.

[32] K. W. Hamlen and M. Jones, “Aspect-oriented in-lined reference monitors,” in *Proc. ACM SIGPLAN Workshop Programming Languages and Analysis for Security (PLAS)*, 2008, pp. 11–20.

[33] I. Aktug, M. Dam, and D. Gurov, “Provably correct runtime monitoring,” in *Proc. Int. Sym. Formal Methods (FM)*, 2008, pp. 262–277.

[34] J. Ligatti, L. Bauer, and D. Walker, “Run-time enforcement of nonsafety policies,” *ACM Trans. Information and System Security (TISSEC)*, vol. 12, no. 3, 2009.

[35] K. W. Hamlen, M. M. Jones, and M. Sridhar, “Aspect-oriented runtime monitor certification,” in *Proc. Int. Conf. Tools and Algorithms for Construction and Analysis Systems (TACAS)*, 2012, pp. 126–140.

[36] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proc. European Conf. Object-Oriented Programming (ECOOP)*, 1997, pp. 220–242.

[37] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of AspectJ,” in *Proc. European Conf. Object-Oriented Programming (ECOOP)*, 2001, pp. 327–353.

[38] United States National Security Agency (NSA), “SELinux,” http://selinuxproject.org.

[39] M. B. Baig, C. Fitzsimons, S. Balasubramanian, R. Sion, and D. Porter, “Cloudtracker: Cloud-wide policy enforcement with real-time VM introspection,” Poster at IEEE Sym. Security & Privacy (S&P), 2013.

[40] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, “A secure environment for untrusted helper applications confining the wily hacker,” in *Proc. USENIX Security Sym.*, 1996.

[41] A. Acharya and M. Raje, “MAPbox: Using parameterized behavior classes to confine untrusted applications,” in *Proc. USENIX Security Sym.*, 2000.

[42] S. N. Chari and P.-C. Cheng, “BlueBoX: A policy-driven, host-based intrusion detection system,” *ACM Trans. Information and Systems Security*, vol. 6, no. 2, pp. 173–200, 2003.

[43] D. Muthukumaran, T. Jaeger, and T. Ganapathy, “Leveraging “choice” to automate authorization hook placement,” in *Proc. ACM Conf. Computer and Communications Security (CCS)*, 2012, pp. 145–156.

[44] D. King, S. Jha, D. Muthukumaran, T. Jaeger, S. Jha, and S. A. Seshia, “Automating security mediation placement,” in *Proc. European Conf. Programming Languages and Systems (ESOP)*, 2010, pp. 327–344.

[45] Ú. Erlingsson and F. B. Schneider, “SASI enforcement of security policies: A retrospective,” in *Proc. New Security Paradigms Workshop (NSPW)*, 1999, pp. 87–95.

[46] F. Chen and G. Roşu, “Java-MOP: A monitoring oriented programming environment for Java,” in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005, pp. 546–550.

[47] K. W. Hamlen, L. Kagal, and M. Kantarcioglu, “Policy enforcement framework for cloud data management,” *IEEE Data Engineering Bulletin (DEB), Special Issue on Security and Privacy in Cloud Computing*, vol. 35, no. 4, pp. 39–45, 2012.

[48] SalesForce.com, “Sales Force,” http://www.salesforce.com, 2013.

[49] Google, “Google Apps,” http://www.google.com/enterprise/apps/business, 2013.

[50] VMWare, “Cloud foundry,” http://en.wikipedia.org/wiki/Cloud_Foundry, 2013.

[51] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE J. Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[52] M. Sun, G. Tan, J. Siefers, B. Zeng, and G. Morrisett, “Bringing Java’s wild native world under control,” *ACM Trans. Information and System Security (TISSEC)*, vol. 16, no. 3, 2013.

[53] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Communications ACM (CACM)*, vol. 51, no. 1, pp. 107–113, 2008.

[54] D. E. Bell and L. J. Lapadula, “Secure computer systems: Mathematical foundations and model,” MITRE Corporation, Tech. Rep. 2547, Vol. 1, 1973.

[55] D. E. Denning, “A lattice model of secure information flow,” *Communications of the ACM (CACM)*, vol. 19, no. 5, pp. 236–243, 1976.

[56] R. S. Sandhu, “Lattice-based access control models,” *IEEE Computer*, vol. 26, no. 11, pp. 9–19, 1993.

[57] N. Swamy, B. J. Corcoran, and M. Hicks, “Fable: A language for enforcing user-defined security policies,” in *Proc. IEEE Sym. Security & Privacy (S&P)*, 2008, pp. 369–383.

[58] J. Mondal and A. Deshpande, “Managing large dynamic graphs efficiently,” in *Proc. ACM SIGMOD Int. Conf. Management of Data (SIGMOD)*, 2012, pp. 145–156.

[59] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, “Securing untusted code via compiler-agnostic binary rewriting,” in *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2012, pp. 299–308.

[60] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *Proc. Sym. Operating Systems Design and Implementation (OSDI)*, 2006, pp. 75–88.