Break it, Fix it: Attack and Defense for “Add-on” Access Control Solutions in Distributed Data Analytics Platforms

Fahad Shaon* 
fahad@datasectech.com 
Data Security Technologies 
Dallas, Texas, USA

Sazzadur Rahaman* 
sazz@cs.arizona.edu 
University of Arizona 
Tucson, Arizona, USA

Murat Kantarcioglu 
murat@datasectech.com 
Data Security Technologies 
Dallas, Texas, USA

Abstract

Distributed data analytics platforms (i.e., Apache Spark, Hadoop) enable cost-effective storage and processing by distributing data and computation to multiple nodes. Since these frameworks’ design was primarily motivated by performance and usability, most were assumed to operate in non-malicious settings. Hence, they allow users to execute arbitrary code to analyze the data. To make the situation worse, they do not support fine-grained access control inherently or offer any plugin mechanism to enable it – which makes them risky to be used in multi-tier organizational settings.

There have been attempts to build “add-on” solutions to enable fine-grained access control for distributed data analytics platforms. In this paper, we show that by knowing the nature of the solution, an attacker can evade the access control by maliciously using the platform-provided APIs. Specifically, we crafted several attack vectors to evade such solutions.

Next, we systematically analyze the threats and potentially risky APIs and propose a two-layered (i.e., proactive and reactive) defense to protect against those attacks. Our proactive security layer utilizes state-of-the-art program analysis to detect potentially malicious user code. The reactive security layer consists of binary integrity checking, instrumentation-based runtime checks, and sandboxed execution. Finally, using this solution, we provide a secure implementation of a new framework-agnostic fine-grained attribute-based access control framework named SECUREDL for Apache Spark.

To the best of our knowledge, this is the first work that provides secure fine-grained attribute-based access control distributed data analytics platforms that allow arbitrary code execution. Performance evaluation showed that the overhead due to added security is low.

1 Introduction

In recent years, the capability of collecting information and its usage is increasing at an exponential rate. Consequently, the big data market is also growing significantly [20]. To process this exorbitant amount of data [17], one of the most popular approaches is to use distributed data processing frameworks [12, 21, 34, 44–46], such as Apache Spark, Hadoop, Hive, Pig, Livy, etc. In this paradigm, typically, data is split and stored over several nodes and a user’s data request is processed in parallel close to the data. These frameworks can be scaled to an increasing amount of data by adding more computing nodes. In these distributed data analytic frameworks, a user can submit custom tasks in different languages to develop custom machine learning or data analytics models over different types of data formats (e.g., JSON, text, image, etc.). Usually, these tasks are written in programming languages such as Java, python, and scala and then compiled to java byte code that are executed on java virtual machines (JVM) on different computing nodes.

Currently, such systems and the data residing on these systems are protected with very basic access control and security mechanisms. In many settings, the user-submitted code is assumed to be secure, and the access control models are based on the underlying distributed file system protections [21,44,45]. For example, an Apache Spark code running to analyze log data stored on Hadoop distributed file system is assumed to be trusted and can see the entire contents of a given file (i.e., access control is enforced only at the file level). Clearly, such coarse-grained access control provided by the default is not enough for many applications. Hence, this limitation spurred further research on enabling fine-grained access control for such systems.

One approach to enable fine-grained access control protections for these systems is by providing higher-level abstractions such as SQL (e.g., Hive built on MapReduce and HDFS, or DeltaLake built on top of Apache Spark). In these approaches, since, user-submitted code is restricted to SQL and the underlying data is relational, existing access control solutions developed for relational databases become directly applicable. However, in the case of arbitrary user-submitted code (e.g., a java code written to build a machine learning model on user log files) working on non-relational data, these solutions would not work directly. For example, if an organization wants a certain data scientist to see logs related to certain errors only, would need to pre-process the data and create a new file that contains these logs only. Therefore, providing access control by creating different versions of the same data source became infeasible as the number of use cases increased.

The lack of plugin support makes it harder to implement a fine-grained access control mechanism for user-submitted jobs with arbitrary code. One approach to sidestep this limitation is to use “add-on” solutions like code instrumentation with inline reference monitors (IRM) [41,42]. IRM is a technique to enforce security policies by injecting security checks into an untrusted code before its execution. For example,
in [41], authors proposed GuardMR to enable fine-grained access control by instrumenting user-submitted jobs to enforce access policies in Hadoop. This and similar instrumentation-based access control enforcement frameworks usually assume that such instrumentation is secure just with the support of the underlying Java virtual machine (JVM) security policies. In this work, we show that such implementations could be attacked, even though JVM-provided sandboxing capabilities are in-place.

More specifically, if users can submit arbitrary code, malicious users can craft a data analytics task to evade access control enforcement by exploiting system APIs, programmatically disabling security managers, runtime code injection, and instrumentation. For example, a user can leverage code injection and instrumentation to evade any security mechanisms by accessing and altering the system’s internal properties. Such code injection can happen on two levels, i.e., (1) loading and using malicious code as a library within the task; (2) loading and using malicious code to alter the platform’s behavior. This raises to question on **how to securely enforce fine-grained access control policies on the user-submitted job with arbitrary codes using instrumentation**.

To address this research question, we systematically analyze threats of API abuse attack and design a two-layered (i.e., proactive and reactive) defense to protect against them with minimal usability and performance overhead. As the name suggests, proactive security is enforced before a user’s request reaches the data framework. On the other hand, reactive security is enforced inside or alongside the data framework. In the proactive part, we utilize state-of-the-art program analysis to detect potentially malicious user code that can be used to evade the instrumented fine-grained access control policies. More specifically, we use static program analysis to screen users’ code against some predefined rules, such as the introduction of framework-specific packages, usage of restricted APIs, use of reflections, arbitrary connection creation, etc. However, some rules in our proactive defense do not guarantee soundness. For them, we use reactive defense as a **safety-net**. Our reactive defense consists of binary integrity checking, static code instrumentation-based runtime checking, and Java security manager. In the binary integrity checking phase, we ensure that all system-specific jars are intact. Runtime checks guard against the cases when an attacker can bypass the proactive defense to use an adversarial coding capability. Finally, the user’s code in our system is executed in a carefully configured sandboxed environment enforced by the Java security manager.

One can argue that since we are creating a sandbox execution environment with the Java security manager, why not enforce every security concern here. Firstly, the security manager cannot prevent all attacks. For example, an attacker can create classes with the system package name, e.g., org.apache.spark. Inside this class, an attacker can access all the package-private methods and fields of other classes in the system packages. Secondly, it does not prevent accessing public properties of a class, which can be leveraged to evade policy enforcement. This suggests that systematic methods combining proactive and reactive defenses are required to prevent all these attack surfaces while enforcing fine-grained access control policies.

Next, we propose a new fine-grained attribute-based access control framework named SECUREDL that uses Scala as the policy specification language to support the enforcement of versatile policies to mask or filter accessed data. Since SECUREDL modifies the user’s submitted job to attach the access control logic to it by leveraging aspect-oriented programming, it is agnostic of any underlying framework. We implemented SECUREDL on top of Apache Spark and Hive. Since Apache Spark supports arbitrary code execution, we adopted our two-layered defense to secure it.

Our contribution can be summarized as follows:

- With a series of concrete attacks, we show that it is possible to evade “add-on” solutions that leverage code instrumentation with inline reference monitors to implement fine-grained access controls atop distributed data analytic solutions. With these attacks, we are the first to show that platform-provided APIs and capabilities are sufficient to evade such solutions.
- We propose a two-layered, proactive, and reactive defense mechanism to protect against these evasion attacks. We show that a combination of static program analysis, sandboxing, and runtime checks can be used to provide protection with low-performance overhead.
- We provide an efficient fine-grained attribute-based access control mechanism for JVM-based distributed data analytics platforms named SECUREDL. To demonstrate its applicability, we integrated it with frameworks with plugin support (i.e., Hive) and without plugin support (i.e., Spark) for fine-grained access control. We adopted our two-layered defense to secure the implementation of Apache Spark.
- Our experimental evaluation showed that our proposed proactive and reactive defense for Apache Spark is effective and incurs low-performance overhead. In a 6 node Hadoop cluster, we observe only about 4% overhead on average on TPCH queries.

## 2 Background and threat model

In this section, first, we provide a background on Apache Hadoop and Spark that would be useful to understand our attacks. Then we briefly discuss the threat model.

### 2.1 Background

**Apache Hadoop architecture.** Apache Hadoop consists of two significant components – a distributed file storage system (HDFS) and a job execution framework (YARN). HDFS splits
files into multiple parts and saves them in different machines. It also creates replicates splits for high availability. The job execution framework executes user-submitted map-reduce code in a distributed manner.

A job in the MapReduce system has a few key components. The user defines the input data format by implementing - InputSplit with details of data chunks, RecordReader containing details of how to read records from these chunks, and InputFormat containing details of how to create InputSplit and RecordReader objects. The user also defines the computation with map-reduce functions, where Hadoop executes the map method once for each record. The map job emits key-value pairs, which Hadoop combines according to the keys and invokes the reduce method once per key.

**Apache Spark architecture.** Like other data processing frameworks, Apache Spark also utilizes the distributed data processing paradigm, where there is a master node (known as *driver*) that receives a data-analytic task and distributes it to various other workers nodes (known as *executors*). In Spark, a user can submit jobs written in a Turing complete language and the system executes the code in a distributed manner. Typically, the Apache Spark cluster operates in two modes, i.e., i) standalone and ii) interactive. In a standalone mode, a user can submit a job jar via SparkSubmit shell to a spark cluster. After job submission, the driver node accepts it, creates a SparkContext within itself, which prepares and sends specific tasks to the executors.

In the interactive mode, users submit code from interactive notebooks (i.e., Zeppelin, Jupyter, etc.), which uses Livy for interactive job execution in a Spark cluster. Livy is an open source REST interface for interacting with Spark. In this setting, Livy acts as a driver and supports executing snippets of code or the entire program. While running from multiple users, Livy relies on user emulations, such as user proxy, to emulate their capabilities.

```java
long count = sc.textFile("users.csv")
.map(line -> line.split(";"))
.map(fields -> Integer.parseInt(fields[1]))
.filter(salary -> salary > 100000)
.count();
```

Listing 1: An example use of Spark RDDs. Here, arrows represent the pointer from a child RDD to its parent RDD.

**RDD, DataSet, and DataFrame.** The main abstraction Spark provides is a data structure named resilient distributed dataset (RDD). RDD abstracts a collection of elements partitioned across the nodes of the cluster and supports a pre-defined set of operations on it, which can be executed in parallel across the cluster. RDD operations are of two types, i) transformations and ii) actions. Transformations create a new RDD from an existing one, and the actions return a value to the driver program after running a computation on the transformed dataset (if a transformation is applied). Typically, initial RDDs are created from files persisted in a distributed file system (e.g., HDFS). Listing 1 presents an example of Spark RDD. Given a file, `users.csv` with user and their salaries, the goal is to find the number of users with a salary of at least 100K. Here, `textFile` creates the initial RDD `map` and `filter` are two transformations. Given an RDD both `map` and `filter` return a new instance of an RDD after applying the transformation defined in the argument. `count` is an action that returns the count of the elements of a given RDD. Spark remembers the transformations by creating a directed acyclic graph (DAG) of all operations. Arrow in Listing 1 represents the parent-child relationship among RDDs in their DAG representation. Similar to RDD, both DataSet and DataFrames are immutable collections of distributed and partitioned data [11].

### 2.2 Threat model

**Attacker Goal:** The goal of our attacks is to evade fine-grained access control on distributed data analytics frameworks with arbitrary code execution capabilities.

**Assumptions:** We consider data lakes in a multiterr organizational setting, where data access is managed by distributed data processing engines and controlled on a need-to-know basis. A data analytic user with lower access privilege can use platform provided capabilities for privilege escalation. We assume that data analytic frameworks are running in a *nix-like environment that enforces file and process level access control and fine-grained access controls atop such frameworks are implemented by leveraging code instrumentation with inline reference monitors (IRM). A malicious user can write arbitrary code to abuse platform APIs (i.e., use of reflection APIs in Java) and network access capabilities. Leveraging other means (i.e., side channels or vulnerabilities of the underlying operating system and non-java libraries) to evade the access control is out of the scope of this work.

### 3 Attacks on IRM-based Approaches

In this section, we present attacks on existing inline reference monitor IRM-based solutions to enforce fine-grained access control mechanisms for distributed data analytic platforms (i.e., Apache Hadoop and Apache Spark). Before we present our attacks, we provide the details of the IRM implementation first for both Hadoop and Spark.

#### 3.1 Attacking IRMs on Hadoop

Vigiles [42] and GuardMR [41] used IRM-based approach to implement fine-grained access control for Hadoop. Since, GuardMR [41] is the most recent work, we will use GuardMR’s implementation to demonstrate our attack, which can be trivially extend for Vigiles. In GuardMR, when users submit jobs with InputFormat, InputSplit, and RecordReader definitions, it constructs a new
InputFormat, InputSplit, and RecordReaders that wraps these methods with policy enforcement. GuardMR uses aspect-oriented programming to implement IRM to detect target methods and inject access control policies into them. Let, the provided untrusted function (e.g., an user-provided RecordReader) is \( f_i \), GuardMR builds a new method \( f_o \) such that \( f_o = f_i(f_e) \). Here, \( f_e \) reads the original data and applies relevant policies, i.e., filters and masks the data, then forwards the data to function \( f_i \).

Scenario #1: Reading with RecordReader. To attack GuardMR, an attacker has to figure out the original data. It boils down to figuring out the files/slits to access by using the Hadoop-provided APIs. It turns out that, inside a provided custom RecordReader attacker can easily read the original input stream and access the data directly, which sidesteps the policies enforced in \( f_o \), which is injected by GuardMR. Note that, such features are not preventable with sandboxing alone, without hurting legitimate functionalities. The attack code snippet is presented in Appendix C.

3.2 Attacking IRMs on Spark

IRMs on Spark. Currently, no solutions exist to implement fine-grained access controls on Apache Spark allowing arbitrary code execution. Hypothetically, to implement such solutions on Apache Spark, the most convenient places are RDD or DataFrame creation methods, such as \texttt{org.apache.spark.SparkContext.textFile(...)}, \texttt{org.apache.spark.sql.DataFrameReader.json(String)}, etc. This is because Spark data users are restricted to use these methods to create an initial RDD and then perform various operations. Like GuardMR, one can inject specialized transformations (i.e., map, filter) to enforce access control on the initial RDD and then return it, so that all the user-defined operations are executed after the policy enforcement. However, a user bypassing the execution of the specialized transformation by retrieving the initial RDD will be able to evade the access control enforcement. Interestingly, each RDD contains an internal reference to its parent RDD (Listing 1) and also to the initial RDD. If an attacker can access these references, she would be able to retrieve the initial RDD before adding new operations. The following attacks will leverage this fact in two different ways.

```scala
val rd = sc.textFile("users.csv")
val clazz = rd.getClass

// #1. Read with "prev" field
val fld = 
    clazz.getDeclaredField("prev")
fld.setAccessible(true)
val parent = fld.get(rd)
val initParent = fld.get(parent)

// #2. Read with "prev" method
val method = clazz.getMethod("prev")
val parent = method.invoke(rd)
val initParent = method.invoke(parent)
```

Listing 2: Retrieving the reference to the initial RDD with Java Reflection to bypass SecureDL.

Scenario #1: Java reflections. To obtain the private properties of an object, one can use reflections. Listing 2 shows a demonstration of retrieving the initial RDD by accessing a private field \texttt{prev} of the RDD which contains the reference to its parent. To provide easy access within the spark framework codes, \texttt{prev} field also has a corresponding package-private method named \texttt{prev}, which can also be used similarly. Some RDDS also have a package-private method \texttt{parent}, which can be used to access any parents, and a convenient method \texttt{firstParent}, which directly returns the reference to the initial RDD.

```scala
val rd = sc.textFile("users.csv")
// accessing the parent pointer
// with "parent" method
val parent = rd.parent(0)
```

Listing 3: Retrieving the reference to the initial RDD with spark specific package naming.

Scenario #2: Spark-specific package. If a user defines a class in a package named "org.spark.", builds the jar and puts it into the classpath. While in execution, there is no distinction between the user’s package and the packages coming from Apache Spark. As a result, a class in the user’s package can access all package-private methods and fields without even requiring reflection. \texttt{firstParent} and \texttt{prev} Methods are package-private, which means these methods are accessible within "org.spark.". However, an attacker can create their class with the same prefix to directly invoke the methods from a Spark job (Listing 3).

3.3 Implementation-agnostic attacks

In this section, we provide two implementation-agnostic attacks on Apache Spark, which can be used to evade any access control mechanism on top of Apache Spark. In these attack scenarios, we assume that a Spark cluster has been set up and a properly configured access control mechanism is in-place.

Scenario #1: Shared executables. The proliferation of insider attacks [4, 9] in large corporations urges the use of controlling access on a need-to-know basis within an organization. Let us assume that Alice and Trudy are two employees in Organization X, where Alice has a higher access privilege...
than Trudy. Alice and Trudy are collaborating on a product recommendation project. This means Trudy can control executables that are run by higher privileged users, which can be leveraged for privilege escalation. The privilege escalation can propagate further up in the chain, an infected user can control executables run by a higher privileged user. Algorithm 1 presents the pseudocode for this attack. While, we implemented the attack on Apache Spark, in theory, such attacks can be implemented on other platforms (i.e., Hadoop) too.

Algorithm 1: Shared Executables Attack on SecureDL

1: procedure INFECTION_JAR(attacker, user_list)
2:     v ← find_victim(attacker, user_list)
3:     jars ← find_shared_jars(a, v)
4:     for jar in jars do
5:         if is_writable(jar) then
6:             user_list ← user_list - {v}
7:         end if
8:     end for
9:     (1) INFECTION_JAR(v, user_list) procedure
10:    (2) Malicious behavior

Scenario #2: Shared notebook server. To support code execution by multiple users, a web-based centralized service, such as Zeppelin, relies on user emulation techniques to emulate different users. More specifically, in the context of Spark, these types of users are called proxy users. A proxy user has the ability to execute the code as any other user in the system. As a result, if a malicious user can successfully modify any executable owned by a proxy user, then they get instant access to the whole cluster. Details of our shared notebook attack can be found in Appendix B.

4 Defenses against arbitrary code execution

Defense goal. By using a similar argument as Fred Cohen [19], it can be proved that given a job J with arbitrary code, detecting if it is malicious is undecidable, in general. The argument holds for all attacks we discussed above. To avoid this pitfall, instead of detection, our goal is to prevent malicious execution without hurting legitimate uses.

Next, we will systematically analyze the adversarial coding capabilities of Apache Spark and show how we can prevent them. Here, Spark represents the group of frameworks (i.e., Spark, Hadoop, Flink, etc.) that support arbitrary code execution. Due to a large number of industry deployments, we focused on Apache Spark. However, a similar method can be adopted for other frameworks too. Note that, this is the first stab towards protecting distributed data analytic systems from a very powerful adversary like this, which might stimulate future research.

4.1 Malicious capabilities in Spark

In this section, we discuss the adversarial coding capabilities of Apache Spark that lead to these attacks. To securely deploy and maintain secure operations of IRM-based access control, these capabilities are needed to be restricted.

1. Preventing reflection on RDDs. Java reflection API allows users to access private properties (field and methods) of an object. Specifically, an attacker can use Java reflection APIs (attack #3) to bypass the SecureDL protection. An intuitive approach to protect against reflection is to sandbox the spark job execution with Java security manager [41]. However, security managers can only protect against access modification and retrieving declared methods or fields. However, it doesn’t guard against invoking public methods. When spark’s internal scala classes are compiled into Java class files, all the package-private methods become public. Because of this, it does not require to perform any access-modification while invoking any of the prev, parent, firstParent methods in Listing 2. Thus, a security-manager-based solution is insufficient and a stronger sandboxing mechanism is required to prevent this attack.

2. Preventing framework-specific package declarations. In attack #4, we see that an attacker can define “org.spark.*” package to directly invoke prev, parent, firstParent methods in scala. Spark jobs must be vetted against such manipulations.

3. Preventing system commands executions. Like other modern programming languages, Java and JVM ecosystem languages (e.g., Scala) allow users to execute system commands and executables. With that privilege, a user can execute any system command and take control of the system. Hence, they must be prevented.

4. Preventing unusual network connections. In Java, users can connect to remote and local processes with a socket connection over the network. A malicious user can actively exploit that to exfiltrate data to a remote server. However, blocking all connections from a submitted task will hamper benign users, since it might block the communication with the namenode from a job as well. So we propose to only block connections that are created from within the user-submitted code.

5. Preventing file read/write. File read/writes enable an attacker to access and temper framework-specific configurations and files (attack scenarios #1 and #2). All the read/writing must be appropriately controlled.

6. Preventing dynamic class loading. Java allows users to load class dynamically, given a class name. This allows the user to load any class in the current classpath. This capability has the potential to enable an attacker to execute non-screened codes including code instrumentation.

7. Preventing to override security managers. A security manager is a class that defines security policies of an application. If it has an implementation of several check* methods such as checkPermission, checkWrite, check-Exec. These methods determine whether particular ac-
tions such as writing a file are permitted in the current running java virtual machine instance. Security managers are typically used to build a sandboxed/protected execution environment. Interestingly if not configured properly, a user can replace an existing security manager, which can be leveraged to bypass the existing protections. To build a secure system, replacing existing security managers and setting up custom policies must be disabled.

8. Preventing native codes and libraries. One can use native codes or libraries to enable any of the features restricted in the Java layer. Hence, loading native libraries and performing native API calls, need to be flagged.

4.2 Defense overview

In this section, we propose a combination of proactive and reactive mechanisms to restrict the adversarial coding capabilities listed in Section 4.1 to ensure an automated secure operation. We would like to emphasize that this paper makes the first attempt to augment IRM-based access control mechanisms with program analysis in a distributed computation environment to improve overall system security.

In theory, we can block all types of code executions and trivially protect the system. However, the question that needs an answer is: “is it possible to guarantee a secure prevention of adversarial capabilities with minimal overheads without denying services to the legitimate users?”

In this section, we answer this question affirmatively. Intuitively, the overhead would be minimal if all adversarial capabilities could be prevented proactively by analyzing the submitted code before execution. However, existing static analysis techniques are known to be unsound (missed detection) and incomplete (generates false alarms) [39]. Missed detections will enable attackers to evade the defense and false alarms will deny services to benign users. To securely block adversarial capabilities with minimal overhead, we employ a combination of proactive and reactive mechanisms. We use a proactive mechanism on two types of capabilities, i) if it can be proactively detected with simple code analysis guaranteeing soundness (no miss detections) with no false alarms, and ii) if it can be proactively detected with a “soundy” (means mostly sound [32], but no guarantees) dataflow analysis framework with fewer false alerts. For the latter case, we also employ a reactive fall-back mechanism to block adversarial uses that evade proactive defense. The goals of the reactive mechanism are as follows, i) ensure a sandboxed execution of the user-submitted code, and ii) block abuse of dangerous APIs that evaded the proactive safety checks.

4.3 Proactive defense

In this section, we present the proactive agent, which uses static code analysis to screen the user-submitted code. Screening is done by checking the code against some well-defined rules. Generally, a rule corresponds to a type of malicious intent. These rules are implemented using static code analysis. Our static code analyzer operates on the submitted code. Some of the rules use regular expressions to find suspicious code. Others use backward data-flow analysis to find influences on a specific program point (e.g., using a dangerous system API), which is used to decide whether the point of interest is used for a malicious purpose.

Identifying code to analyze. Identifying the code that is controlled by a malicious user is instrumental to provide seamless service to the legitimate users. For example, various third-party libraries use Java Reflection APIs to offer convenient utility. If the code analysis engine blindly rejects a job with Java Reflection API invocation, a legitimate user using such libraries will be impacted. To solve this problem, we offer library whitelisting service. Our code analysis engine will skip the screening of a jar or class binary if it is whitelisted. We created a list of common libraries that are whitelisted by default. The list can be extended or modified by an administrator. In addition, to help with the whitelisting processes, admin can run SecureDL’s program analysis on the input jar file. We also provide interfaces to search in common public sources, such as GitHub, the maven repository, CVE database. Finally, the program analysis results of a library can be shared among organizations. To whitelist a jar, first we compute the hash of the jar and store it. Then we unzip it and compute the hashes of each of the class files and store them in a database.

During the static analysis of a jar, the analyzer first creates a hash of the jar and looks up the database to see whether it exists in the whitelist or not. If found, then the analysis engine skips it. Otherwise, it unzips the jar and creates hashes for each of the class files from inside the jar. If the hash of a corresponding class is not found, then the class is included in the static analysis, otherwise it is skipped.

To improve the performance, our analysis engine can also maintain a cache of the analyzed code. If the analysis result of a jar or a class is available in the analysis cache, then we could retrieve the analysis result from the cache and skip the reanalysis. Next, we discuss the use of static analysis to implement our proactive defense.

i) Sound detection with static analysis. Here, we present the cases where static analysis can be leveraged for sound and (almost) complete detection.

Detection of framework-specific packages. For security purposes, Apache Spark intentionally put some of the framework internal APIs as package-private, so that these APIs are hidden from the external users. As we discussed in Section 4.1, a user can define classes with the framework-specific package structure with a prefix of “org.apache.spark”, so that the framework internal APIs become accessible. A naive approach would be to reject a user-submitted job, if it has a package prefix “org.apache.spark”. However, several machine-learning libraries benignly use this capability for optimizing job performance. Therefore, to prevent the adversarial use of these capabilities, we only block jobs that leverage this capability to invoke the APIs to access the parent objects of
Figure 1: Blocking the use of reflection on RDD objects. Here, $\mathcal{V}_i \rightarrow p_\sigma$ represents an influence of an object $\mathcal{V}_i$ on the program point $p_\sigma$. $\mathcal{V}$ represents the set of all such objects.

an RDD.

Detection of restricted APIs. We also restrict users to invoke the following system APIs (1) to load classes dynamically; (2) to override the security manager; (3) to execute system commands; (4) to create remote connections with SSL/TLS APIs; (5) using native codes/libraries, (6) standard file read/write APIs, etc. One might argue that an attacker could use third-party libraries that are not covered by our defense to perform these operations. However, in such cases, the attacker is also needed to embed the libraries within her submitted, which would use the standard APIs we cover. We proactively block the remote connection and file read APIs from the user-submitted jar because these functionalities are heavily used by Apache Spark, blocking them with security managers will incur significant performance overheads.

It is trivial to show that if a user cannot load and execute codes dynamically, the above detection mechanisms will be able to guarantee the soundness for these cases. Note that this does not guarantee completeness (no false alarms), i.e., a benign user might also need to use these APIs legitimately. However, such use cases are rare in practice. During our evaluation, we did not encounter any.

ii) Soundy detection with static analysis. For these cases, our proactive agent does not guarantee soundness. We rewrite the user-submitted Spark jobs to ensure runtime checks for the cases that are missed during our proactive phase. If misuse is detected, our runtime fallback mechanism blocks further execution, which guarantees the security of the framework.

Detection of reflection API invocations. Java Reflection offers the mechanism to modify the behavior of methods, classes, and interfaces at runtime. In Section 4.1, we observe that to obtain the private properties of an object, we need to invoke `java.lang.Object.get(java.lang.Object)` on the corresponding field by using the object of interest as the parameter. Similarly, to invoke methods on an object, it is required to invoke `java.lang.Object.invoke(java.lang.Object, java.lang.Object[])` on the corresponding method by using the object of interest as the first parameter. To block accessing the parent object of an RDD, it is sufficient to prevent passing an RDD as a parameter to these reflection methods. We use backward data-flow analysis to detect these cases. Specifically, our backward data-flow analysis identifies whether an RDD instance passes as an input parameter to these methods, which is formally defined in Figure 1.

**Backward dataflow analysis implementation.** Since implementing a new dataflow analysis is not our main focus, we use the interprocedural backward data-flow implementation of CryptoGuard [39] for this purpose. CryptoGuard’s implementation is demand-driven and known to produce fewer false alarms, which is suitable for our case. However, like all the other static dataflow analysis frameworks, it does not guarantee soundness. Note that to improve the performance, CryptoGuard can be replaced with any other competing solutions [31, 40], which is beyond the scope of this work.

4.4 Reactive defense

Our reactive defense has two types of components - i) a static component, which checks the integrity of all framework components before running a job, and ii) a dynamic component, which works as a fall back for some of the proactive defenses and blocks some of the capabilities that are not covered by the proactive defense (i.e., blocking suppressAccessChecks with reflections). The dynamic component consists of two parts (1) Java security manager-based sandboxing, and (2) rewriting the user-submitted jar with runtime checks. Using security managers is the most natural way to sandbox untrusted code execution in JVM. However, the existing Java security manager capabilities are insufficient to sandbox some of the exploitable capabilities. Because of this, in addition to the security manager, we also use code instrumentation-based runtime checking. The implementation details of our reactive defense can be found in Appendix D.

5 Framework-agnostic access control

In this section, we present a new framework-agnostic fine-grained access control named SECURED for distributed data analytic frameworks. Although there have been some efforts to define access control models for big data analytics systems (e.g., GuardMR [41]), compared to previous work that is specific to certain environments such as Hadoop, we define our access control policies agnostic to Spark like distributed data analytics platforms. In addition, compared to previous work, in our policy definition, we separate the filtering and obfuscating (masking) aspects, since the underlying execution environment such as Spark, enables us to write specialized filter and map tasks.

We represent the input data as a dataframe. Dataframe is a structured data storage that stores data in rows and columns. Formally, an input data set $D_{id} = \langle id(D), C, T \rangle$ consists of an unique identifier (e.g., filename, table name), column definitions, and an ordered set of tuples. This abstraction is
very powerful and encompasses a wide variety of data types. Intuitively, a dataframe directly maps to a relational table. In addition, we can represent any non-relational data using this abstraction. For example, this abstraction can be used to model a text file where we assume each line is a tuple of a single element and can assume the name of the element is ‘text’. In addition, we can model arbitrarily nested Json data, where each attribute of the input Json becomes an element of the tuple. In reality, a wide variety of popular data analytics systems represent data in this format, such as Spark [11], Pandas [37], R [23], etc. Furthermore, we can represent non-textual, such as images, data into dataframes by keeping a column of binary data (BLOB in a relational database). This simplifies data processing since we can efficiently manage meta-data as well.

Policy in our system defined as \( P = (I, A, M, f) \), where \( I(D_id, u, A) \) is a boolean function for deciding whether a given dataframe \( D_id \), a user \( u \), and set of attributes \( A \), the policy is applicable or not. \( M(t, u, c) \) is set of masking functions, \( f(t, u, c) \) is user provided boolean function for limiting view of the data applied to each tuple \( t \in D_id \) using user information \( u \) and system context information \( c \) (e.g., IP address of the request).

We use the dataframe matching function \( I \) during user request processing to determine whether a dataframe \( D_id \) has any policies attached to it. This function may use any attribute of the dataframe, a set of attributes \( A \), and the user attributes \( u \) as desired by the function \( I \). For a given policy \( P \), a set of attributes \( A \) can be used to represent any user attribute such as user role (e.g., user John is a data scientist) or user information (e.g., user Jane is working at location X). The same policy can be applied to multiple dataframes. In our system, given a specific user \( u \), and specific dataframe \( D_id \), if multiple policies are activated, we raise an alarm and deny request until the conflict is solved by the policy admin. Alternatively, the system can be configured to apply the first matching policy. A filter \( f(t, u, c) \) in our system is a user-provided boolean function that has access to the input tuple \( t \), accessing user \( u \), and connection context information \( c \). Data owners in our system can write the filter in scala. This allows users to create policies that are as complex as the users want.

A masking or obfuscation function \( m \) in our system takes input of a type of data, modifies it, and then returns same type of data with limited information (i.e. \( \text{type}(a) = \text{type}(m(a)) \)). Let \( X(\text{regex}, s) \) be a function that takes a regular expression and a string value and returns the indexes of string regular expression matches, \( S(\text{matches}, s, \text{pattern}) \) be a substitution function that takes the regular expression matches, original string, and a pattern, return the string with replaced pattern in matching location. For example, a regular expression-based US phone number masking function that only returns the last four digits can be expressed as

\[
\text{index} = X(\text{\textasciitilde}d3)?(-)d3-d4', s) \\
m_p(s) = S(\text{index}, s, \text{\textasciitilde} d4' - ddd')
\]

**Algorithm 2 Policy Enforcement**

**Ensure:** \( I(D_id, u, A) = \text{True} \) \( \triangleright \) Ensure that policy is applicable

1: \( \text{procedure POLICY ENFORCEMENT}(P, u, c, D_id) \)
2: \( \triangleright \) Apply policy \( P \) for a request for dataframe \( D_id \)
3: submitted by user \( u \) given the request context \( c \)
4: \( D' \leftarrow \emptyset \)
5: \( \text{for all } t \in D_id \text{ do} \)
6: \( \text{if } f(t, u, c) = \text{True then} \)
7: \( D' \leftarrow D' \cup M(t, a, m) \)
8: \( \text{Return } D' \)

For efficiency reasons, we define masking functions specific to a column. Let \( M(t, a, m) \) be a column specific masking function that applies masking function \( m \) on column \( a \) of tuple \( t \), i.e. \( M(t, c, m) = m(t.a) \). Finally, in \( M \) we have ordered sets of masking functions potentially for each different column, \( M = \{M_1, M_2, \ldots\} \).

In summary, given a policy \( P \), user \( u \) and date frame \( D_id \), first the system checks whether \( I(D_id, u, A) \) returns true (i.e., Line 0 in Algorithm 2). If that is the case, for each tuple \( t \in D_id \), it checks whether \( f(t, u, c) \) returns true (i.e., Line 5 in Algorithm 2). Then for all \( t \in D_id : f(t, u, c) = \text{True} \), it adds the masked version of the tuple \( t \) to the resulting data frame (i.e., Line 6 in Algorithm 2). Since our system allows arbitrary scala code for functions \( f, m \), it can represent any existing role-based (RBAC) [27] and attribute-based access control policies (ABAC) [29].

Since our system allows us to specify ABAC policies using the Scala programming language, any user and data attributes can be combined with programming languages to enforce very sophisticated security and privacy policies. For example, using a custom-defined function defined on images, a policy that can redact human faces automatically can be defined in our system. In other words, a mask function \( M \) defined over images can use an ML subroutine to detect the human faces and replace the detected pixels with black ones to redact human faces. We would like to stress that our policies are generic enough to represent any ABAC policies defined on the dataframe abstraction. As we discussed above, this abstraction can represent policies at any granularity for relational, semi-structured, and unstructured data.

**5.1 Implementation using AOP in Spark**

While implementing SECUREDL atop Apache Spark, our goal is to keep the enforcement system as transparent as possible from the data user’s point of view, i.e., without introducing
new APIs. All existing jobs written using current API calls must work in our new system. To implement the fine-grained access control in this manner, we have two options - (1) we could rewrite the distributed data analytics system with the necessary enforcement codes, and build our version of Spark (i.e., embed the reference monitor inside the system), (2) use an inline reference monitor (IRM) (i.e. we attach our enforcement logic at run-time) [25].

For our system, we chose the IRM approach because changing and rebuilding existing systems is difficult and time-consuming. Simply, given a policy \( P \), user-submitted job \( j \), our policy rewriter will rewrite the job \( j \) into \( j' \) so that the policy is enforced. For a policy \( P \), it maps masking operations with a map transformation and filter operations with a filter transformation. To implement IRM-based policy enforcement in our system, we choose Aspect-oriented programming (AOP). We defer the discussion of implementation details to Appendix A. In Section 3, we discussed several concrete attacks that can evade our IRM-based implementation in Spark. We use the proactive and reactive defenses discussed in Section 4.2 to defend against evasion attacks discussed in Section 3.

Figure 2: System overview of attribute-based policy enforcement in Apache Spark with proactive and reactive defenses. Here proactive agents, reactive agents, policy dispatcher and filter caching are the new components proposed in SecureDL.

**System overview** In this section, we provide an overview of the whole system with an attribute-based access control framework and defense in place for Apache Spark. Figure 2 shows the system overview. In this system setup, data analytics users can submit tasks through SparkSubmit client and the interactive Zeppelin server*. Admin users define attribute-based policies and send them to the policy dispatcher. Policy dispatcher maps the policies into map and reduce transformations. It bundles all data masking operations into a map and arbitrary data filtration logics into a callback method, which can be invoked from a filter. Then, it sends the maps into Spark drivers and the callback method binary to a distributed

\* Apache Zeppelin is a "Web-based notebook that enables data-driven, interactive data analytics and collaborative documents with SQL, Scala and more" [1]

Figure 3: Overhead of running different TPCH queries on 100GB data in Hive with SecureDL enabled.

We trivially implement our SecureDL in Hive by leveraging Hive’s plugin system for fine-grained access control. Specifically, we wrote an authorizer class by extending the public interface `org.apache.hadoop.hive.ql.security.authorization.plugin.HiveAuthorizerFactory` to integrate our access control checking logic. In addition, we implemented a factory class extending `org.apache.hadoop.hive.ql.security.authorization.plugin.HiveAuthorizerFactory`. In this class, we instantiate the authorizer class with proper parameters. Finally, to configure hive authorization process properly we set configuration variable `hive.security.authorization.manager` to `true` and `hive.security.authorization.manager` to full classpath of our authorizer class.

To test the overhead of our hive reactive enforcement, we load 100GB of TPCH data on Hive and execute TPCH queries 1 to 5. In Figure 3 we show the overheads. We observe that overheads range from 0.88% to 23.94%. This wide range in overhead is due to the fact that a few queries contain operations on policy-controlled columns and others do not.

5.2 Implementation with Hive Plugins

We trivially implement our SecureDL in Hive by leveraging Hive’s plugin system for fine-grained access control. Specifically, we wrote an authorizer class by extending the public interface `org.apache.hadoop.hive.ql.security.authorization.plugin.HiveAuthorizerFactory`. In this class, we instantiate the authorizer class with proper parameters. Finally, to configure hive authorization process properly we set configuration variable `hive.security.authorization.manager` to `true` and `hive.security.authorization.manager` to full classpath of our authorizer class.

To test the overhead of our hive reactive enforcement, we load 100GB of TPCH data on Hive and execute TPCH queries 1 to 5. In Figure 3 we show the overheads. We observe that overheads range from 0.88% to 23.94%. This wide range in overhead is due to the fact that a few queries contain operations on policy-controlled columns and others do not.

Figure 3: Overhead of running different TPCH queries on 100GB data in Hive with SecureDL enabled.
6 Evaluation

We performed extensive experiments to quantify the overhead of different components in Apache Spark when our fine-grained access control and defense mechanism is in place. In this section, we present our experimental results.

Cluster configurations. We ran experiments on Hadoop Spark clusters with one master node, a few worker nodes, and one service node. The master node hosts the resource manager and name nodes of Yarn and HDFS, the worker nodes, host node managers and data nodes, the service node hosts Kerberos, LDAP, policy manager, and other support services. All these nodes are running inside a virtual cloud network, which is located in a cloud availability zone. We ran our experiments in Oracle Cloud Infrastructure (OCI) and each node in the cluster is of type VM.Standard2.4 having 4 OCPU, 60GB of main memory, running Ubuntu 18.04 OS. We also mount a block device disk of size 1TB on each instance. We are using Hadoop version 3.3.0, Spark 3.0.1, and Livy 0.8.0 snapshot (HEAD 4d8a912).

Spark and HDFS configurations. In our setup, the HDFS data directories, such as dfs.datanode.data.dir, dfs.namenode.name.dir, hadoop.tmp.dir are pointed to the directories in the mounted block device. For simplicity, we keep the replication factor 1. In this setup, we need on average 1 min 53 sec to copy a single file of size 1GB from local disk to HDFS with hadoop fs -copyFromLocal command. In addition, we also configured memory and virtual cores for Yarn and Spark-based on the number of nodes in the cluster and per node available resources. We defer the detailed discussion to Appendix E.

6.1 Performance of static components

In this section, we present the performance analysis of the static components of our system that do not depend on the dataset or computation. Our proactive defense, runtime jar instrumentation with AspectJ, and jar integrity checking are such components. Specifically, our analysis will answer the following questions.

• What is the overall overhead of the static components?
• What is the accuracy of proactive defense? Does it block legitimate cases?

Proactive defense overheads. To compare the overhead of our proactive defense with baseline, we run the Livy server with and without proactive defenses. We collected 55 Scala code snippets from official Apache Spark examples. These examples include code for reading and writing different types of files, performing machine learning, and analyzing data. The average line of code in the snippets is 21.78. We run 10 iterations of Scala code execution with and without proactive defense. To analyze Scala snippets proactively, it goes through three distinct phases - (1) code generation where we generate a compilable sbt project out of user-submitted code with proper dependencies, (2) code compilation, where we compile the code with sbt, (3) proactive analysis, where we perform our code analysis on the generated binaries. Code generation and compilation depend on disk performance and code analysis depends on the complexity of the logic. In our experimental evaluation, we observed that the compilation step is the most expensive. In Table 1, we show the overhead of proactive analysis. In all cases, we see an approximately constant overhead for code analysis.

| Setup                          | Avg. Time (#55 Scala Snippets) |
|-------------------------------|--------------------------------|
| Livy w/o Proactive Enforcement| 10.470 (s)                     |
| Livy w/ Proactive Enforcement | 17.666 (s)                     |

Table 1: Proactive overhead on 55 Scala snippets in Livy.
In summary, we observe an almost constant overhead in instrumenting and jar integrity checking, which was expected.

**Accuracy of proactive defense.** To check the soundness and precision of our proactive analysis on realworld code, we collected 200 Apache Spark projects from GitHub. We specifically searched with keywords 'spark tutorial', 'spark learning', etc. and picked top repositories. We determined the build system by checking for standard build files such as pom.xml, build.gradle. We observed that a majority of the projects are not maintained and we failed to compile due to dependencies. We successfully built 33 projects in total, on which we run our proactive analysis. In 4 cases, our analysis ran out of memory. On average our analysis took 5.61sec per jar file. In 29 projects, the most prevalent violation is reading data from outside servers (4 cases). For example, accessing a CSV file from a public S3 bucket for analysis purposes.

From the security stand point, blocking such cases might be acceptable in an organizational setting. To extensively evaluate the overhead associated with the size of the dataset, we generate and load TPCH data, and run queries for phone number masking. With security manager enabled we observe mean overhead of 11.44% with standard deviation 5.43%. Finally, phone and comment masking together, we observe similar average overhead 15.03% with standard deviation 4.48%. For query 2, apart from few extreme cases, we observe overhead around 16%. Now, for query 6 we observe mean overhead of 2.21% with standard deviation 1.42% for phone masking. With security manager enabled we observe mean overhead of 2.54% with standard deviation 2.56%. Finally with phone and comment masking we observe mean overhead of 1.15% with standard deviation 0.94%. For query 6 our overhead is in and around 2%. For query 14, we observe average overhead of 3.72% with standard deviation 1.4% with phone masking. With security manager we observe average overhead 3.33% with standard deviation 1.24%. Finally with phone and comment masking we observe mean overhead 2.54% with standard deviation 2.56%. Similar to query 6 we observe overall overheads of around 2% in query 2. To summarize, we do not observe any variation in runtime overhead over the baseline with the increase of the dataset size. However, the overhead is highly dependent on the type of query and the policies we enforce. In our case, the query 2 has complex inner query, several joins, and order by clauses, resulting noticeable overheads. In contrast, query 6 and 14 has complex aggregation and simpler joins, so the overheads are significantly lower.

### Impact of the dataset size on overheads.

In this experiment, we generate and load 10GB, 20GB, 30GB, 40GB, and 50GB of TPCH data in HDFS and run queries 2, 6, and 14 on them. To measure the runtime overhead, we used the following four settings, i) without any access control policy enforcement, ii) with phone masking policies, iii) with phone and comment masking policies, iv) with phone masking and security manager enabled.

In Figure 5 we illustrate the execution time of these queries. For query 2 (Figure 5(a)) we observe average overhead of 12.85% with standard deviation 8.17% for phone number masking. With security manager enabled we observe mean overhead of 11.44% with standard deviation 5.43%. Finally, phone and comment masking together, we observe similar average overhead 15.03% with standard deviation 4.48%. For query 2, apart from few extreme cases, we observe overhead around 16%. Now, for query 6 we observe mean overhead of 2.21% with standard deviation 1.42% for phone masking. With security manager enabled we observe mean overhead of 2.54% with standard deviation 2.56%. Finally with phone and comment masking we observe mean overhead of 1.15% with standard deviation 0.94%. For query 6 our overhead is in and around 2%. For query 14, we observe average overhead of 3.72% with standard deviation 1.4% with phone masking. With security manager we observe average overhead 3.33% with standard deviation 1.24%. Finally with phone and comment masking we observe mean overhead 2.54% with standard deviation 2.56%. Similar to query 6 we observe overall overheads of around 2% in query 2. To summarize, we do not observe any variation in runtime overhead over the baseline with the increase of the dataset size. However, the overhead is highly dependent on the type of query and the policies we enforce. In our case, the query 2 has complex inner query, several joins, and order by clauses, resulting noticeable overheads. In contrast, query 6 and 14 has complex aggregation and simpler joins, so the overheads are significantly lower.

**6.2 Overheads of dynamic components**

Our access control mechanism and reactive defenses are enforced at runtime, which depends on the dataset and the nature of the computation. In this section, we perform several experiments to extensively evaluate the overhead associated with them. Our experimental evaluation answers the following research questions.

- How does the overhead of our attribute-based access control and reactive defense change over the baseline with the size of the dataset?
- What is the impact of the number of computing nodes?

**Experimental setup.** For this experiment, we use TPCH benchmark

```
http://www.tpc.org/tpch/
```

We run TPCH queries on CSV data using Spark. More specifically, we store the TPCH tbl tables in HDFS as CSV files, load them as dataframes in Spark, and run the TPCH queries on them. For these experiments, we set up two sets of policies.

1. **Masking on phone columns.** We show the last 4 digits of 12 digits on the phone column of customer and supplier table of TPCH.

2. **Masking on comments columns.** We use regular expressions to detect phone numbers and email address inside the comments column of all the tables of TPCH and replace with defined patterns. We use regular expression `\d\d\d\d` to detect phone numbers and replace with `***-***-dddd` pattern, where d represents a digit in the input string. This masking essentially shows only last 4 digits of the phone number. Similarly for email addresses, we use the regular expression `\b[\w\s]+@[a-zA-Z0-9-]+[a-zA-Z0-9-]*\d?([a-zA-Z0-9-]+[a-zA-Z0-9-]*[a-zA-Z0-9-]*\d+\d?\d?\d?|\d+\d?\d?\d?)\b.`

   * to mask emails in the form of *@*c, to only show the last character of the email address.

Specific details of the policies are listed in Appendix F.
increased computation capacity increases the parallelization in the cluster, hence, the overhead of our map and filter execution decreases. At some point, the overhead of parallelism, i.e. network and io overhead of exchanging data among nodes, will diminish this computational overhead. Furthermore, a very important point to emphasize is that Spark always greedily allocates all available memory, whether it is needed or not. For query 2 on phone masking, we observe that overhead increases 1% to 9% for phone masking with increasing capacity. We observe a similar pattern for the combination of phone and comment masking as well.

### 7 Discussion

**Comparison with others.** Currently, our policy enforcement mechanism is unique, hence we do not have any meaningful benchmarks to compare against. With our job modification approach, we can enforce complex policy on non-structured data such as text or image. For example, we can mask phone number or email address using regular expression on text file dynamically depending on the policy. To the best of our knowledge, this capability is unique among Spark providers. Finally and most importantly, our proactive component is complimentary to existing policy enforcement of Spark distributions and policy enforcement.

**Applicability to other systems.** We demonstrated that our fine-grained access control mechanism can be integrated with frameworks irrespective of plugin supports. With plugin support, its easy to integrate SecureDL. Without plugin support, the main challenge is to guarantee a secure integration. In theory, the core concept of two-layered defense system can be applied to other similar distributed computation frameworks.
Apache Spark, which can support policy specification for ABAC policies to enforce access control for the Apache Spark Access control methodologies for data management systems (i.e., Hadoop, Flint, etc.) that run on JVM ecosystem. To do that, one needs to analyze the corresponding platform APIs to identify their adversarial coding capabilities and then carefully design the detection mechanism so that it minimally affects the usability and performance.

**Soundness and accuracy.** During our evaluation on 55 Scala code snippets, our proactive defense did not raise any alerts. Our evaluation on 29 real-world projects also indicates a low possibility of blocking legitimate uses. However, if our system raises a false alert for a jar or class file, to ensure an uninterrupted operation, a system admin can whitelist it and exclude it from analysis. Our manual analysis on real-world projects did not discover any new suspicious features that our analysis might have missed. To further evaluate the detection capabilities, we created a set of 15 attacks including all the attack scenarios discussed in Section 3.2 and Section 3.3. Evaluation shows that our proactive analysis successfully detected all the cases. Guarding against the adversarial use of reflection APIs through static dataflow analysis is the unsound part of our system. Our jar re-write based runtime checks theoretically guarantee their prevention.

## 8 Related work

**Access control methodologies for data management systems.** Access control methodologies have been applied to data management systems over the years ranging from relational databases (e.g., see discussion and references in [14]) to NoSQL systems such as Hadoop (e.g., [13, 22, 41]) based on two major access control model. In the Role-based access control [27], each user is assigned to roles, and roles are assigned to permissions on the underlying data management resources such as tables, etc. Recently, attribute-based access control methodologies (ABAC) [29] have been suggested by NIST. In ABAC, the attributes of the users (e.g., role information or the location information of a user), data objects combined, and context information (e.g., IP address of the data access request) can be used to define access control policies.

Some prior research works try to apply ABAC in big data management systems. For example, in [28], ABAC is applied in the context of Hadoop services, and ABAC policies are used to give access to entire services and data objects such as Hive tables or columns. In SparkXS [38], authors propose using ABAC policies to enforce access control for the Apache Spark Streaming engine. The proposed methodology provides access to entire streams in an all-or-nothing fashion (i.e., no capability for sanitizing data on the fly) through REST endpoints and does not allow user code execution. SmartGuard [10] also provides ABAC policy enforcement in Apache Spark for relational data only.

Compared to all these work, **SECUREDL** is the only fine-grained attribute-based access control framework with data masking and sanitization for distributed environments like Apache Spark, which can support policy specification for any type of data ranging from text to image in addition to structured/relational data. In other words, not only do our policies decide what part of the data (irrespective of the data source type) a user can access or not but also decide the granularity of the data (e.g., replacing the first 6 digits of SSN with ‘*’). To our knowledge, none of the existing work implemented ABAC with masking extension in the context of distributed big data systems such as Spark.

Another relevant data access control mechanism is purpose-aware access control (PAAC), where we define and enforce access control policies based on the purpose of the computation. PAAC and ABAC are complementary to each other. GuardSpark++ [43] implements PAAC in Spark SQL for structured data (SecureDL operates on both structured and non-structured data). It utilizes Spark’s internal SQL query optimization engine ‘Catalyst’ and enforces purpose-aware policies between analysis and optimization states.

**Commercial solutions.** There are several commercial open-source projects (i.e., Apache Ranger) that offer access control atop distributed data analytics platforms. Intrinsically, the capability of Ranger is limited by the capability granted by the plugin system of the host framework. Since Apache Spark does not have a fine-grained access control plugin system, Ranger can not support it directly. All large cloud vendors have their own version of data analytics and access control mechanisms. For example, an Amazon Web Services customer can load CSV data in an s3 object store and run a HiveQL query using a service named Athena [2]. Access control in this scenario will be equivalent to the access control settings in the underlying data store s3. In our view, these tools solve only a part of the problem. These solutions revolve around solving the access control on structured data. In contrast, SecureDL can work with any type of data and support much more complex access control policies with more advanced proactive protection. Furthermore, a user of these systems can incorporate components, such as SecureDL proactive analysis, into their existing workflow easily.

**Static code analysis for vulnerability detection.** Static code analysis has been extensively used to detect API misuse vulnerabilities Java code [15, 16, 24, 26, 31, 33, 35, 36, 39, 47]. Most of the work focuses on detecting system-level API misuses [15, 24, 26, 31, 35, 39], such as SSL/TLS [26, 39], Cryptographic APIs [24, 31, 39], APIs for fingerprint protection [15], Android Inter-app communication APIs [16], etc. Some of the recent works focus on non-system APIs too [33, 47], such as cloud service APIs for information storage [47], Creditcard information processing APIs [33], etc. In this scenario, no missed detection is expected-but-not-critical. In our case, a missed detection has a serious consequence on the overall security guarantee. Consequently, we employ runtime checks to detect and block such cases.
9 Conclusion

In this work, we proposed a new fine-grained attribute-based access control mechanism named SecureDL for distributed data analytics frameworks. If plugin support is not provided, we use aspect-oriented programming to inject access checking logic at runtime. We guarantee its enforcement with a two-layered defense mechanism under arbitrary code execution. We are the first to utilize the program analysis to complement a prototype implementation. Our extensive experimental evaluation shows that the SecureDL has a low overhead while securely enforcing attribute-based access control policies.

References

[1] Apache Zeppelin. https://zeppelin.apache.org/. Accessed August 16, 2021.
[2] AWS Athena. https://aws.amazon.com/athena. Accessed August 16, 2021.
[3] HiBench is a big data benchmark suite. https://github.com/Intel-bigdata/HiBench. Accessed August 16, 2021.
[4] Insider threats present big risk to financial institutions. https://www.securitymagazine.com/articles/93904-insider-threats-present-big-risk-to-financial-institutions. Accessed August 16, 2021.
[5] Javassist - java bytecode engineering toolkit since 1999. https://www.javassist.org/. Accessed August 16, 2021.
[6] Permissions in the java development kit (jdk). https://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html. Accessed August 16, 2021.
[7] Redis access control list. https://redis.io/topics/acl. Accessed August 16, 2021.
[8] Redis is an open source (bsd licensed), in-memory data structure store, used as a database, cache, and message broker. https://redis.io/. Accessed August 16, 2021.
[9] Rise of darknet stokes fear of the insider. https://krebsonsecurity.com/2016/06/rise-of-darknet-stokes-fear-of-the-insider/. Accessed August 16, 2021.
[10] Smartguard. https://www.axiomatics.com/news/axiomatics-launches-smartguard-for-data-spark-sql-edition/. Accessed August 19, 2021.
[11] Spark sql, dataframes and datasets guide. http://spark.apache.org/docs/3.1.1/sql-programming-guide.html. Accessed August 16, 2021.
[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, pages 1383–1394, 2015.
[13] Feras M Awaysheh, Mamoun Alazab, Maanak Gupta, Tomás F Pena, and José C Cabaleiro. Next-generation big data federation access control: A reference model. Future Generation Computer Systems, 108:726–741, 2020.
[14] Elisa Bertino, Gabriel Ghinita, and Ashish Kamra. Access control for databases: Concepts and systems. Found. Trends Databases, 3(1–2):1–148, January 2011.
[15] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, and Wenke Lee. Broken Fingers: On the Usage of the Fingerprint API in Android. In 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 21-24, 2018.
[16] Amiagnshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In ACM AsiaCCS’17, pages 71–85, 2017.
[17] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. Mob. Networks Appl., 19(2):171–209, 2014.
[18] Cloudera. Determining hdp memory configuration settings. https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.6.4/bk_command-line-installation/content/determine-hdp-memory-config.html. Accessed August 16, 2021.
[19] Fred Cohen. Computer viruses: Theory and experiments. Comput. Secur., 6(1):22–35, 1987.
[20] Louis Columbus. 10 charts that will change your perspective of big data’s growth. https://www.forbes.com/sites/louiscolumbus/2018/05/23/10-charts-that-will-change-your-perspective-of-big-datas-growth/. Accessed August 16, 2021.
[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004, pages 137–150, 2004.
[22] Philip Derbeko, Shlomi Dolev, Ehud Gudes, and Shan-tanu Sharma. Security and privacy aspects in mapreduce on clouds: A survey. Computer science review, 20:1–28, 2016.
[23] R Documentation. Data frames. https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/data.frame. Accessed August 16, 2021.
[24] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In ACM
Maanak Gupta, Farhan Patwa, and Ravi Sandhu. An
Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R
David F. Ferraiolo, Ravi Sandhu, Serban Gavrila,
Sascha Fahl, Marian Harbach, Thomas Muders,
Úlfar Erlingsson.
Aravind Machiry, Chad Spensky, Jake Corina, Nick
Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden,
Robert Huie. Properly sizing workloads
Xiangrui Meng, Joseph K. Bradley, Burak Yavuz,
Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen,
Doris Xin, Reynold Xin, Michael J. Franklin, Reza
Zadeh, Matei Zaharia, and Ameet Talwalkar. Millib:
Machine learning in apache spark. J. Mach. Learn. Res.,
17:34:1–34:7, 2016.
Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan
Zhang, Donglai Zhu, and Min Yang. Finding Clues
for Your Secrets: Semantics-Driven, Learning-Based
Privacy Discovery in Mobile Apps. In 25th Annual
Network and Distributed System Security Symposium,
NDSS 2018, San Diego, California, USA, February 18-
21, 2018, 2018.
Marten Oltrogge, Nicolas Huaman, Sabrina Amft,
Yasemin Acar, Michael Backes, and Sascha Fahl. Why
eve and mallory still love android: Revisiting {TLS} (in)security in android applications. In 30th {USENIX}
Security Symposium ( {USENIX} Security 21), 2021.
Pandas. Pandas data frame. https:
//pandas.pydata.org/pandas-docs/stable/index.html#pandas.
Accessed August 16, 2021.
Davy Preuveneers and Wouter Joosen. SparkXS: ef-
cient access control for intelligent and large-scale
streaming data applications. In 2015 International
Conference on Intelligent Environments, pages 96–103,
2015.
Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad
Shao, Ke Tian, Miles Frantz, Murat Kantarcioglu, and
Danfeng Yao. Cryptoguard: High precision detection
of cryptographic vulnerabilities in massive-sized java
projects. In Proceedings of the 2019 ACM SIGSAC
Conference on Computer and Communications Security,
pages 2455–2472, 2019.
Johannes Späth, Karim Ali, and Eric Bodden. Ide:
efficient and precise alias-aware dataflow analysis. Proc.
ACM Program. Lang., 1(OOPSLA):99:1–99:27, 2017.
Huseyn Ulusoy, Pietro Colombo, Elena Ferrari, Mu-
rat Kantarcioglu, and Erman Pattuk. Guardmr: Fine-
grained security policy enforcement for mapreduce sys-
tems. In Proceedings of the 10th ACM Symposium on
Information, Computer and Communications Security,
ASIA CCS ’15, Singapore, April 14-17, 2015.
Huseyn Ulusoy, Murat Kantarcioglu, Erman Pattuk, and
Kevin W. Hamlen. Vgilex: Fine-grained access control for
mapreduce systems. In 2014 IEEE International
Congress on Big Data, Anchorage, AK, USA, June 27-
July 2, 2014, pages 40–47. IEEE Computer Society,
2014.
Tao Xue, Yu Wen, Bo Luo, Boyao Zhang, Yang
Zheng, Yanfei Hu, Yingjiu Li, Gang Li, and Dan Meng.
GuardSpark++: fine-grained purpose-aware access con-
trol for secure data sharing and analysis in spark. In An-
nual Computer Security Applications Conference, AC-
SAC ’20, page 582–596, New York, NY, USA, 2020.
Association for Computing Machinery.
A Spark AOP details

We create pointcut for all the user-facing methods in Spark that are used to read files and create RDD or DataFrame, such as `org.apache.spark.SparkContext.textFile(String, int)`, `org.apache.spark.sql.DataFrameReader.json(String)`, etc. An Spark data user can utilize these methods to create an initial RDD and then perform various operations. We find these methods by searching statements that instantiate RDD and its subclasses in the Spark source code. We then look for public methods that use these methods. We attach AspectJ javaagent to all Spark’s Java processes by modifying the shell script, `spark-class`. In addition, to communicate with other services, such as policy service, we need to initialize some service clients in spark context. So, we implement and attach a SparkListener. Specifically, we modify these five configurations - `spark.driver.extraJavaOptions`, `spark.executor.extraJavaOptions`, `spark.driver.extraClassPath`, `spark.executor.extraClassPath`, `spark.executor.extraListeners`

We use `@Around()` and related annotations from AspectJ to attach policy enforcement code to data reading methods. One such example is listed in Listing 4. Here we define a method `policiesOnTextFile` with an `@Around("execution(* org.apache.spark.SparkContext.textFile(String, int))")` annotation, which signals AspectJ to attach the `policiesOnTextFile` method around `SparkContext.textFile` method. In other words, any time the spark context method is executed we first receive the call in our `policiesOnTextFile` method. This method take as argument `ProceedingJoinPoint` class and return either a modified RDD with access control enforcement or raise error due to insufficient access permission. We get all the original input parameters, such as the file path, using `getArgs` method of the joint point class. Next, we decide whether the user has access to the file using file metadata information. If the user has access to the file we execute the `proceed` on the joint point. This creates the initial RDD or DataFrame. To emphasize, this does not actually execute the read access, rather creates a job in a DAG that will be executed later. Now we modify this RDD or DataFrame with access control policy implementation.

```python
@Around("execution(* org.apache.spark.SparkContext.textFile(String, int))")
def policiesOnTextFile(joinPoint):
    file_path <- joinPoint.getArgs[0]
    u <- fetch_user_info()
    if (!hasAccess(u, file_path)) {
        throw new AccessControlException()
    }
    rdd <- joinPoint.proceed()
    return enforce_policies(file_path, rdd)
```

Listing 4: SecureDL advice with point cut using AspectJ annotation

In the policy enforcement method, we fetch policies and user information from our central policy server. Then we collect connection information, such as the user’s IP address. Next, we serialize and distribute the policies. In particular, we create executable byte code of the filters and masks in the matching policies and distribute the executables by using a central distributed cache server. Finally, we attach a filter and a map method with the input DataFrame or RDD. In the filter method, we execute the serialized filter method from the matching policies and in the map method, we execute data masking policies. In Listing 5 we outline the implementation of the policy enforcement method.

```python
def enforce_policies(file_path, rdd):
p <- fetch_policies(file_path)
u <- fetch_user_info()
c <- connection_info()
f, m <- serialize_and_distribute(p)
rdd.filter(t -> f(t, u, c)) .map(t -> m(t, u, c))
return rdd
```

Listing 5: SecureDL policy enforcement implementation
The example code of Listing 1 with policy enforcement will have two more modification methods a filter and a map just after reading the file as listed in Listing 6. Although we did not list explicitly here, we implemented similar enforcement for all available DataFrame and RDD creation methods. To summarize, we attach access control enforcement policies using APO. We attach our advice to spark’s data reading methods and ensure these get executed by modifying the appropriate spark parameters in the spark execution script.

```java
long count = sc.textFile("users.csv")
    .filter(t -> f(t, u, c))
    .map(t -> m(t, u, c))
    .map(line -> line.split(";"))
    .map(fields ->
        Integer.parseInt(fields[1]))
    .filter(salary -> salary > 1000000)
    .count()
```

Listing 6: An example of applying access control before executing any user defined transformations

**Implementation completeness.** One of the biggest challenges in our implementation is ensuring that we are trapping all methods. Otherwise, an attacker can bypass the security mechanism by reading data using those methods. Therefore, to complete our implementation, we examined all available official tutorials and thoroughly went over the source codes of related packages in Spark for the listed methods of reading data in Spark. Furthermore, if new data reading methods are introduced later, we can easily write ‘advices’ for these methods. However, we observe that data reading method changes are infrequent. Apache Spark tends to keep the user-facing API consistent over version updates. Therefore, machine learning models written in one version can run on a different version without further modification.

**B Details of our shared notebook attack**

Zeppelin uses daemon processes called interpreters to execute user code in the underlying framework. We observe that some interpreters execute code in the zeppelin server. For example, Spark interpreter creates a local spark context in the Zeppelin server. Any code submitted in this spark context is executed with the privilege of the Zeppelin server’s Unix user. Now, if the Zeppelin server has a working Livy interpreter, the Zeppelin unix user must have access to the Livy configurations including Kerberos keytabs, and URL. This is required to properly communicate with the Livy server. As mentioned earlier, Zeppelin uses proxy users or super users in Livy to emulate other users. Now, an attacker can create a spark session using the Spark interpreter and execute the code to i) read Livy configuration for the proxy user to communicate with the Livy server, ii) create a Livy session on behalf of a victim user by utilizing the proxy user infrastructure. We outlined this attack scenario in Algorithm 3. Here, an attacker starts an interpreter with the necessary jars, scans for Livy configurations, creates a Livy session by impersonating a victim user, and runs the analysis code to access data. In addition, once an attacker gets access to an interpreter that runs an arbitrary code as Zeppelin unix user, it can rewrite the jars to persist this attack. Because the Zeppelin unix user has write access to all internal and interpreter’s jar files.

**Algorithm 3**

```
1: procedure EXECUTE_CODE_IN_LIVY(uname, code)
2:   Start interpreters with necessary jars
3:   Execute the following procedures in Spark interpreter:
4:   c ← read_livy_credentials()
5:   lc ← create_livy_client(c)
6:   s ← lc.create_session(uname)
7:   s.execute(code)
```

**C Attack on GuardMR**

Following is the code snippet that evades GuardMR protection on Apache Hadoop.

```java
class MalReader extends RecordReader {
  public void initialize (...) {
    List<FileSplit> splits =
        fileInputFormat.getSplits(job);
    for(FileSplit split : splits) {
      final FutureDataInputStreamBuilder builder =
          file.getFileSystem(job)
              .openFile(split.getPath());
      FSDataInputStream fileIn =
          FutureIOSupport.awaitFuture(builder.build());
      long start = split.getStart();
      long end = start + split.getLength();
      int length = 1024 * 1024;
      byte[] buffer = new byte[length];
      long position = start;
      while (position < end) {
        ...
      }
    }
  }
```

Defense with security managers. JVM ecosystem offers security managers to secure sandbox untrusted code. Given the context (call trace with invocation parameters) of a system call invocation, security managers can block its execution (by throwing exceptions), if the operation is not permissible. Permission represents access to a system resource. The list of permissions that can be checked by using security managers can be found here [6]. To prevent some of the adversarial capabilities, we use security managers in the following manner.

- **Blocking to set system security policy.** Our implemented security managers block the ability to set new security managers and new policies to the existing security managers from the user-submitted code. These permissions have the potential to enable attackers to override existing security manager capabilities.

- **Blocking reflections.** Security managers allow to block the following reflection permission, i) accessDeclaredMembers – querying public, protected, private properties of a class, ii) suppressAccessChecks – accessing public, protected, private properties of a class, and iii) newProxyInPackage – creating proxy instances of a nonpublic interface in a given package. By checking the invocation parameters, we block all these permissions if they are used to access RDD properties. This effectively blocks the attack #1 in Listing 2. However, an attacker does not require to use any of these reflection capabilities to invoke a public method of an RDD via reflection (Attacks 2, 3, 4 in Listing 2), which is still open. To prevent these attack surfaces, we use instrumentation-based runtime checking.

- **Blocking code executions and write.** To be on the safe side, using security managers, we also block executing system commands. By analyzing the context of file write operations we selectively block this feature. For example, writing to the system or Spark-specific directories are disabled, so that an attacker cannot modify framework-specific contents. We do not block the file write capability entirely. Blocking file read operations by security managers would significantly slow down the system. This is because Spark’s normal operations are heavily dependent on this feature. As a result, Invoking security managers to check the execution context for file read operation will incur significant performance overheads. We rely on the proactive defense to block any file reads with java.io or java.nio APIs.

### Selective read/write with runtime instrumentation.

We proactively block all read/write through standard Java APIs. However, an attacker can still perform read/write operations with Apache Spark’s standard APIs. Since we already instrument these APIs through Aspect-Oriented Programming (AOP) for access control, we add additional checks to prevent any suspicious read/writes.

#### Defense with instrumentation-based runtime checks.

In the first part of this section, we see that the security manager-based defense is not sufficient to block some of the adversarial capabilities that could be missed by the proactive defense, e.g., accessing public property using reflection. java.lang.Object.invoke(java.lang.Object,java.lang.Object[]) is an example of such an API. To guard against this, we introduce a runtime check just before the invocation of this API. If it is invoked on an instance of RDD or a sub-class of RDD, we generate a runtime exception. We employ the same checks for java.lang.Object.get(java.lang.Object) API.

### E Memory Calculation

To allocate resources for the YARN container, we adopt the technique outlined in [18]. We exclusively consider two resources - memory and virtual cores (vCores). We reserve some memory and vCores for system processes, then divide the remaining memory into containers. We use number of containers $\alpha$ and memory per container $\beta$ as input to calculate the total memory allocation available for node manager, minimum and maximum memory allocation limit, application manager allocation limit, map-reduce memory allocations, and vCores allocation limit. In our setup, we calculate the number of vCore per node by multiplying 2.5 to available OCPU, since the OCI OCPU are not shared with other tenants [30]. So, for a VM.Standard2.4 node calculate number of available vCores is $10 = 4 \times 2.5$. We reserve 2 vCores and 12GB of memory for system processes. That leaves us with 8 vCores and 48GB of memory for yarn container in a node. We set minimum container resources to 1 vCores and 6GB. We use the equations outlined in Table 2 to calculate all relevant memory configurations for yarn. We also need to tune resources for Apache Spark. In particular, we need to divide the available vCores and memory into executors. Given $\gamma$ vCores in a worker node, $\delta$ vCores per executor, $\epsilon$ executors per worker and $\omega$ workers in total, we can calculate the number of executors per node as $\epsilon = \left\lfloor \frac{\gamma}{\delta} \right\rfloor$. Multiplying this value with the number of worker nodes gives us the total available executors. We reserve one executor for resource ne-
gotiation. For memory allocation per executor, we divide the total available memory per node by the number of executors per node. In our setup, we decided to go with 2 vCores per executor setup. So, on a 4 worker nodes cluster the relevant parameters are -executor-cores 2 -num-executors 15, and -executor-memory 10752MB.

| Configuration                                      | Equation                       |
|----------------------------------------------------|--------------------------------|
| yarn.nodemanager.resource.memory-mb                | $= \alpha \cdot \beta$        |
| yarn.schedulerm.minimum-allocation-mb              | $= \beta$                      |
| yarn.schedulerm.maximum-allocation-mb              | $= \alpha \cdot \beta$        |
| mapreduce.map.memory.mb                            | $= 2 \cdot \beta$              |
| mapreduce.reduce.memory.mb                         | $= 0.8 \cdot \beta$            |
| mapreduce.reduce.java.opts                         | $= 0.8 \cdot 2 \cdot \beta$    |
| yarn.app.mapreduce.am.resource.mb                  | $= 0.8 \cdot 2 \cdot \beta$    |
| yarn.app.mapreduce.am.command-opts                 | $= 0.8 \cdot 2 \cdot \beta$    |

Table 2: Yarn and Spark resource calculation formulas

F Policies

Masks:
- phone:
  - name: PhoneNumberMask
type: regex_mask
detection_regex: "^\(?\d{3}\)?(-|\s)\d{3}-\d{4}$"replacement_pattern: '***-***-dddd'

- email:
  - name: EmailMask
type: regex_mask
data_type: email
detection_regex: "^[a-zA-Z0-9]+@\[a-zA-Z0-9-._]+\.[a-zA-Z]{2,3}$"replacement_pattern: '********@

Executors per node $\epsilon = \lfloor \frac{\gamma}{j} \rfloor$
Number of executors $= \epsilon \cdot \omega - 1$
Executor memory $= \frac{\alpha \cdot \beta}{k}$

14of12d:
- type: static_mask
data_type: digit
length: 12
name: ShowLast4of12Digits
visible_anchor: end
visible_chars: 4

Policy:
- customer_accounts:
  - document: customers.accounts
  - filter: |
    val ip : String
    context("ip").asInstanceOf[String]
    val z : Integer
    row("zip").asInstanceOf[Integer]
    if (ip == "10.5.17.19") {
      // Zeppelin IP
      z == 75080
    } else if (ip == "10.5.17.10") {
      // Command line IP
      z >= 75080 \&\& z <= 75081
    } else {
      false
    }

Listing 8: Policy example