The Queen's Guard: A Secure Enforcement of Fine-grained Access Control In Distributed Data Analytics Platforms - SecureDL

Fahad Shaon *^ - DataSecTech
Sazzadur Rahaman* - University of Arizona
Murat Kantarcioglu - DataSecTech

* Contributed equally
^ Current affiliation Google

Agenda

- Opportunities and contribution
- Threat model
- Background - Apache Spark
- IRM based access control - "add-on" security
- Attacks on IRM based solution
- Defense against these attacks
  - Proactive and reactive
- Evaluation results

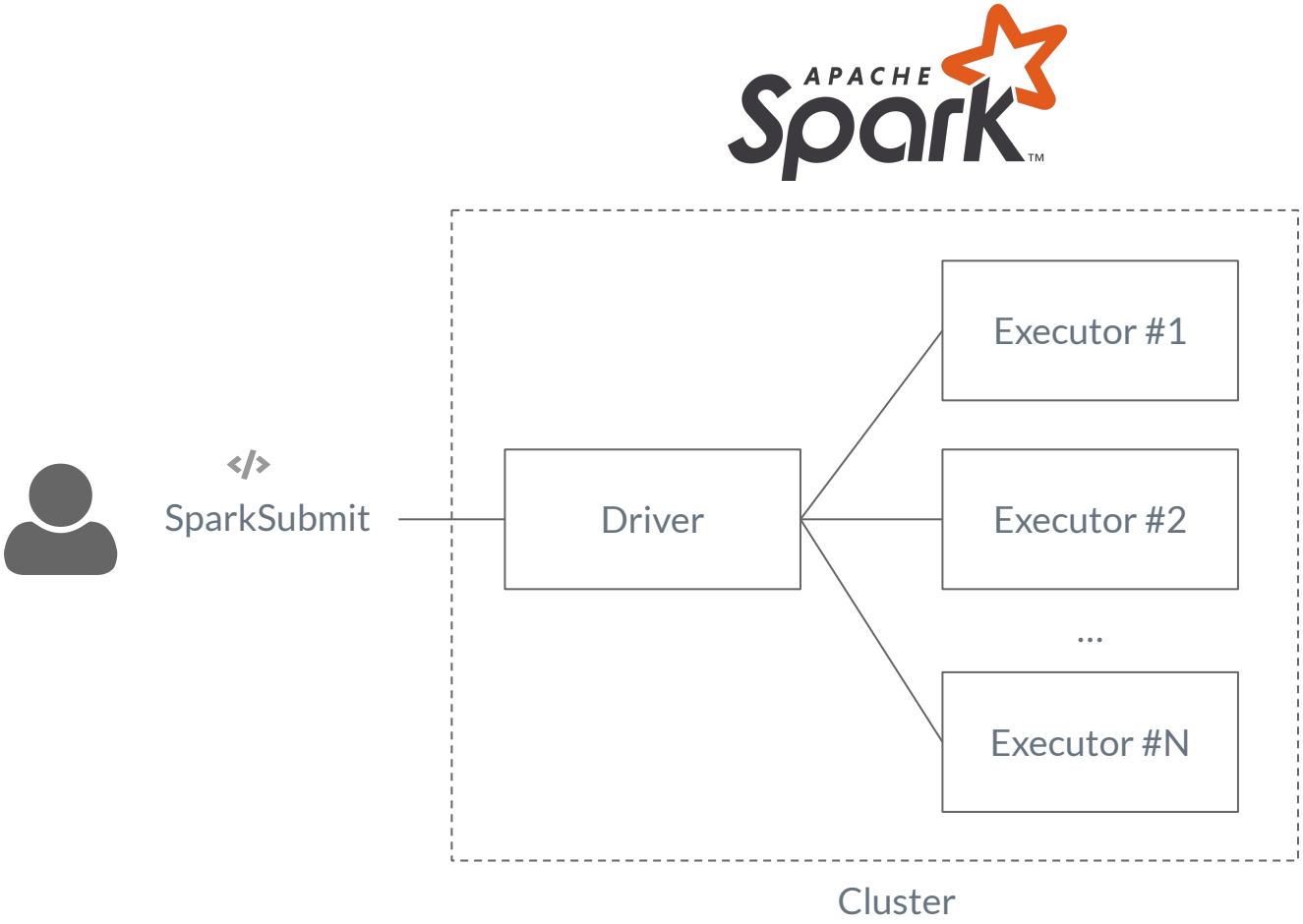
Opportunities and Contribution

- Apache Spark
  - Doesn't have built in fine-grained security
- "add-on" security solutions are inadequate
  - We show attacks using invasive system API
- Propose two layer defense mechanisms
- Propose fully customizable access control with masks and filters

Threat Model

- Attacker's aim: Evade fine-grained
- Is an **insider** in multi-tier organization
  - *Has* lower privilege
  - *Can* run code for data-analytics
- Has incentive to evade ACL
  - Especially if chance of getting caught is low
- Real world use cases
  - Criminals Increasing SIM Swap Schemes to Steal Millions of Dollars from US Public
  - Spotlight on Insider Fraud in the Financial Services Industry

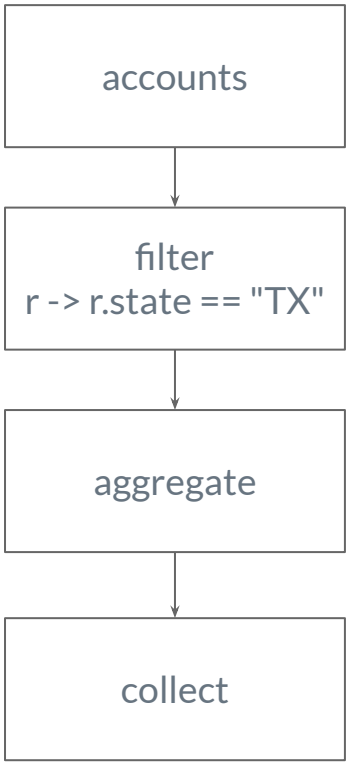
Background - Spark Job Execution



Background - Spark Programming

```
spark.read.json("accounts.json")
 .filter(r -> r.state == "TX")
 .groupBy("zip").agg(mean("rewards"))
 .collect()
```

Program to read a json file, filter rows, and aggregate



RDD

- After code submission the code gets translated into RDD (Resilient Distributed Dataset)
- It is lazy evaluated, until necessary computation won't happen

Access Control using AOP / IRM

```
@Around("execution(* org.apache.spark.sql.DataFrameReader.json(...))")
def policisOnJsonFile(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, u, rdd)
```

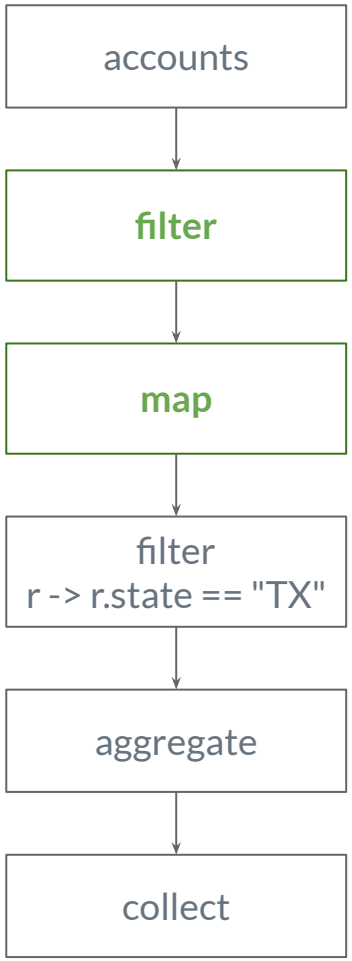
Check user has access to the file

Execute the JSON reading function

Add a map() and a filter() functions to add data masks and filters based on admin defined policies

Aspect Oriented Programming based implementation to enforce policy on json file

RDD after policy enforcement



RDD

- After the policy enforcement the RDD have additional filter and map
- **Filter** function is used to remove rows that user doesn't have access to
  - e.g. *User1* don't have access to accounts with zip *75080*
- **Map** function is used for modifying content of a data
  - e.g. mask all but last 4 digits of credit card
- Added filter and map gets distributed
- Similar to GuardMR, Vigiles

Policy - Encoded in Yaml

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

   l4of12d:
     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")
     val z : Integer = row("zip")

     if(ip == "10.5.17.10") {
       z >= 75080 \&\& z <= 75081
     } else {
        false
     }

   masks:
     credit_card:
       - Masks.l4of12d
     comments:
       - Masks.phone
```

Attack Surfaces on IRM based Solution

```
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)
```

```
// #3. Read  with "parent" method
val mthd = clazz.getMethod("parent", 0)
val initParent = mthd.invoke(rd, ...)

// #4. Read with "firstParent" method
val method =
clazz.getMethod("firstParent")
val initParent = method.invoke(rd, ...)

// Accessing the parent pointer
// with "parent" method
val parent = rdd.parent(0)
```

Apache Spark - Attack Surfaces

- Restricting reflection on RDDs.
- Preventing framework-specific package declarations.
- Preventing dynamic class loading.
- Preventing to override security managers.
- Preventing native codes and libraries.

SecureDL - System Architecture

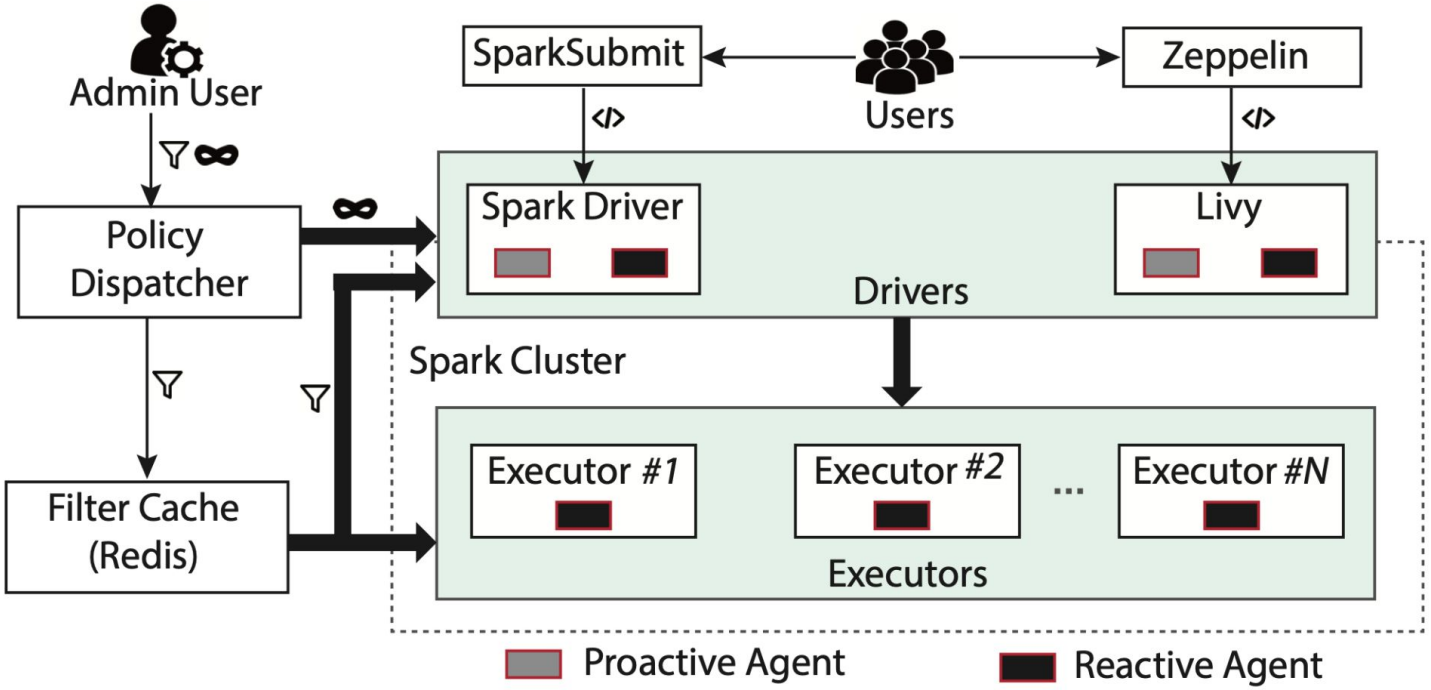


Figure 2: System overview of our policy enforcement in Apache Spark with proactive and reactive defenses. Here proactive agents, reactive agents, policy dispatchers, and filter caching are the new components proposed in SECUREDL.

Defense - Proactive

Use program analysis (**static analysis**) to block

- Framework specific package declaration
- Restrict permissive System API
  - Dynamic classloading
  - Security Manager overriding
  - Native code/library loading
- These can be invasive in some cases
  - Implemented allowlisting mechanism

Defense - Proactive

- Use program analysis (**backward dataflow**) to detect reflection API usages
  - Track use of *java.lang.Object get(java.lang.Object)* and *java.lang.Object invoke(java.lang.Object,java.lang.Object[])*
  - Especifically if RDD instance is first parameter to this
  - Note: JavaSecurityManager can't protect against *get* or *invoke* calls
- Utilized [CryptoGuard]

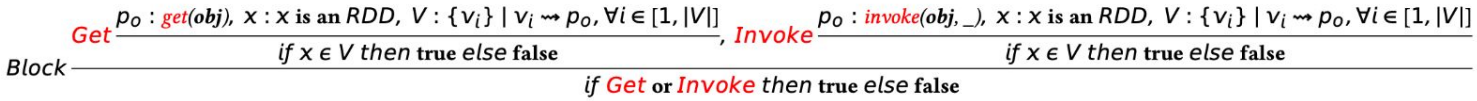


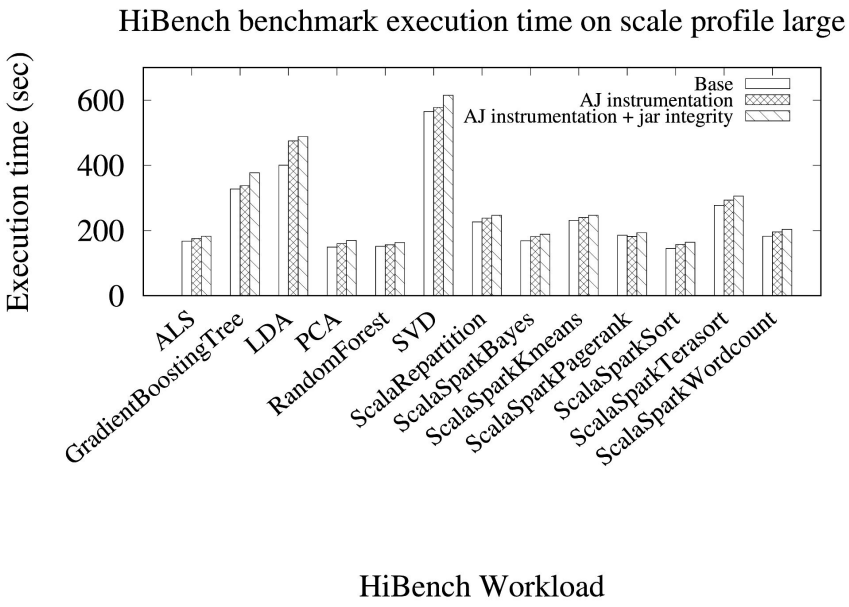
Figure 1: Blocking the use of reflection on RDD objects. Here, $v_i \rightsquigarrow p_o$ represents an influence of an object v_i on the program point p_o. V represents the set of all such objects.

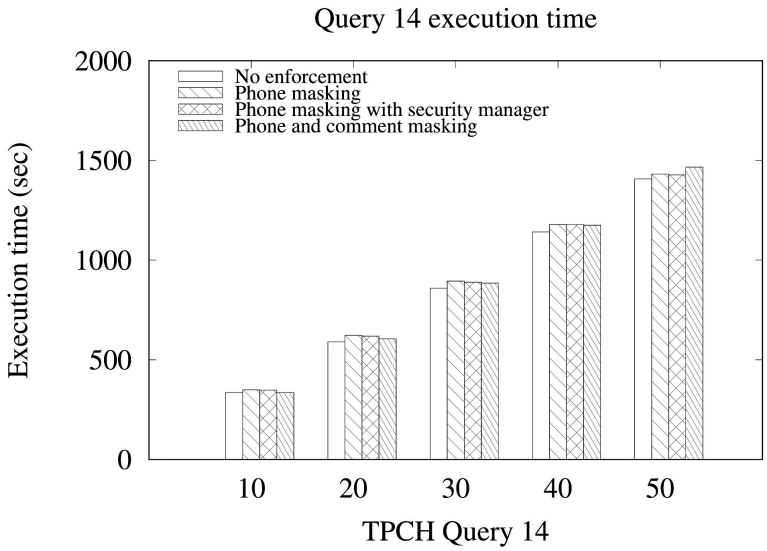
Defense - Reactive

- Enable a Security Manager that restricts method calls
  - `accessDeclaredMembers`
  - `suppressAccessChecks`
  - `newProxyInPackage`
- Analyze call trace to
  - find if a call generated from user submitted code

Evaluation - Access Control Overhead

RQ: What is the overhead of policy enforcement?





- Policy overhead is highly policy dependent
- Average overhead 4% on TPCH query with masking policy
- Paper contains many more experimental results

Evaluation - Proactive Analyzer

RQ: What are the common proactively detectable issues in spark programs in the wild?

- Collected 2120 spark repositories from GitHub
- 637 were built using maven
- 417 were successfully built
- Found 247 analyzable jars
  - Exclude uber-jars
- Found some issues in 21 jars
  - 12 jars had `org.apache` package
  - 7 jars use `Class.forName`
  - 8 jars has networking calls

Questions?

Contacts

Fahad Shaon - *fs@shaon.dev*
Sazzadur Rahaman - *sazz@cs.arizona.edu*
Murat Kantarcioglu - *murat@datasectech.com*

OSS - *https://github.com/DataSecTech*