

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

Presented by Gaudiano Antonio

Scope of the presented work



In recent years the need for big data analysis increased exponentially and so the various distributed data processing frameworks.



Most of them use very basic access control systems, often enforced only at the file level.



Can fine-grained access control be implemented via higher-level abstractions? Can external «add-on» solutions be effective? How about sandboxing?

Technologies' background

Apache Hadoop:

- YARN
- HDFS
- MapReduce

Apache Spark:

- Driver – Executors
- Resilient Distributed Dataset

Threat Model



Attacker Goal: evade fine-grained access control without leaving any traces (transient attack) by abusing the platform-provided APIs.



Assumptions:

Attacker is an insider, a low privileged user that can run code for data-analytics purposes.

Already present security systems relies on Inline Reference Monitors (IRM).

IRM

- An Inline Reference Monitor (IRM) is a security mechanism that enforces access control policies by modifying or inserting code directly into a program (or system) at runtime.

```
// original code
String data = readFile("/user/data/sensitive.txt");

// IRM-modified code
if (hasAccess(user, "/user/data/sensitive.txt")) {
    String data = readFile("/user/data/sensitive.txt");
} else {
    throw new SecurityException("Unauthorized access");
}
```

Attacking IRMs on Hadoop

Hadoop internally uses GuardMR for IRM which wraps user-submitted jobs with policy enforcements.

An attacker could exploit a custom RecordReader (which GuardMR expects and therefore does not check) and, instead of using the safe methods, manually open a file stream using the known file path and offset, then read the raw bytes directly!

Hadoop RecordReader bypass

```
class MalReader extends RecordReader {  
  
    public void initialize (...) {  
  
        List<FileSplit> splits = (List<FileSplit>) 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) {  
                position += fileIn.readBytes (position, buffer, 0, length )  
                // Access the plain text values  
            }  
        }  
    }  
}
```


Attacking IRMs on Spark

Spark currently lacks fine-grained access control mechanisms.

One idea could be applying specialized transformations at RDD / DataFrame creation to enforce policies before user operations...

But attackers can bypass this by retrieving the initial unmodified RDD since they holds a reference to its parent.

First Scenario: Java reflections

- Powerful feature that allows a program to inspect and modify its own structure and behaviour at runtime.
- Often used by attackers to bypass 'private' or 'protected' access modifiers.
- So the idea of the attack is to use reflections in order to access private methods of RDDs to get its original unmodified state.

Spark RDD manipulations with Reflections


```
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, ...)
```

Second Scenario: Abusing Package Namespace

- Exploit Java's package visibility rules to gain unauthorized access to sensitive internal informations.
- The idea is to create a malicious class within a legitimate Spark package so that the JVM sees that class as being part of the same package as Spark's internal code.

Spark IRM bypass with Package Spoofing

```
package org.apache.spark;  
val rd = sc.textFile("users.csv")  
// accessing parent pointer  
val parent = rd.parent(0)
```



Defense strategy

- Is it possible to guarantee a secure prevention of adversarial capabilities with minimal overheads without denying services to legitimate users?
- Blockable Attack Surfaces:
 - program's behaviours that do not appear in normal, legitimate use cases.
- Non-Blockable Attack Surfaces:
 - APIs used by both malicious and benign code, like reflection.



Proactive defense

Based on static code analysis to screen user-submitted code.

Regular expression to check method's invocation.

Backward data-flow analysis.

Library allowlist service based on hashes.

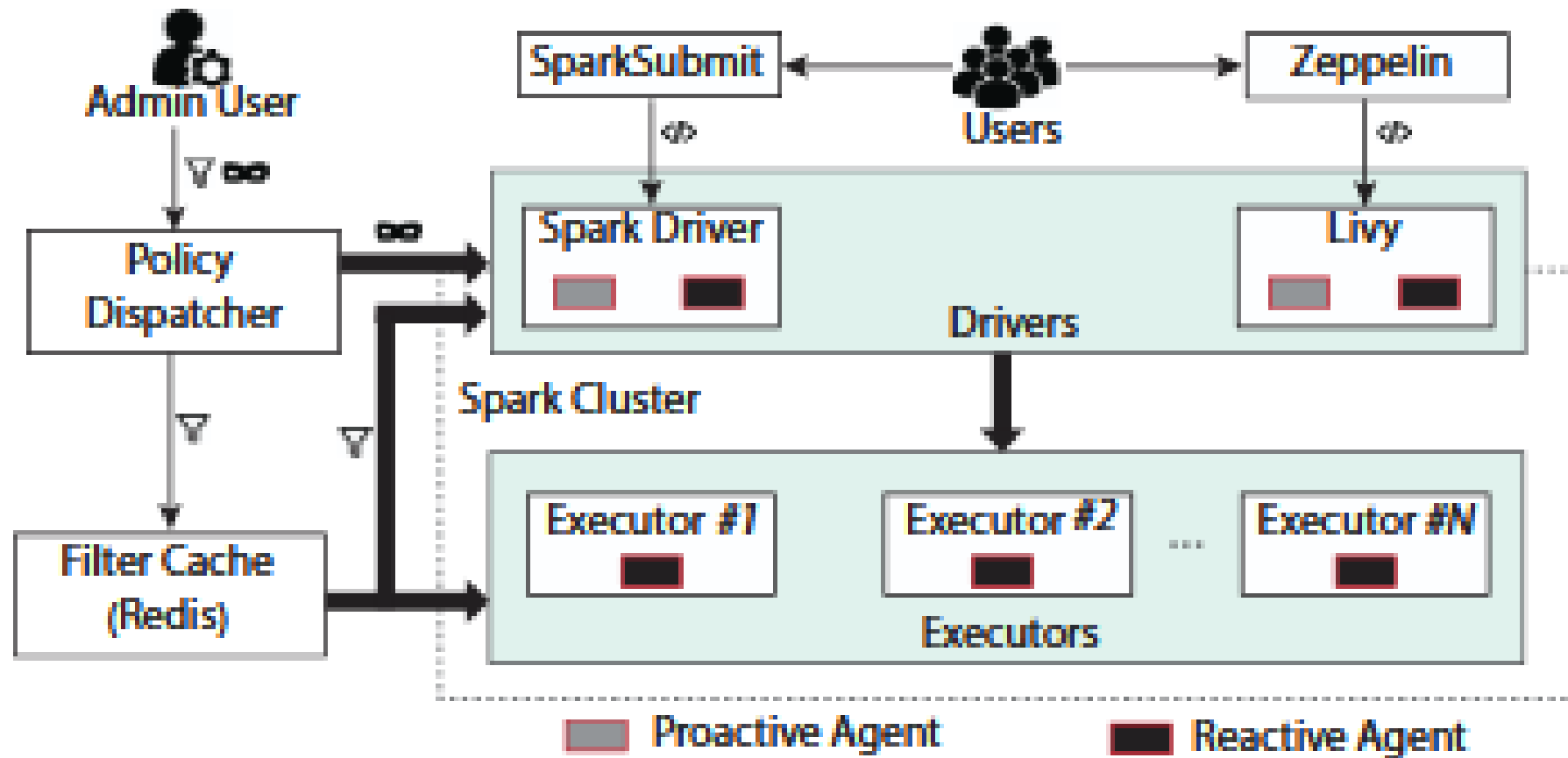
Reactive defense

Restricts non-blockable attack surfaces that escaped or not covered by the proactive analysis.

Exploits Java sandboxing capabilities to detect abuse cases.

If needed , rewrites user-submitted jobs with runtime checks

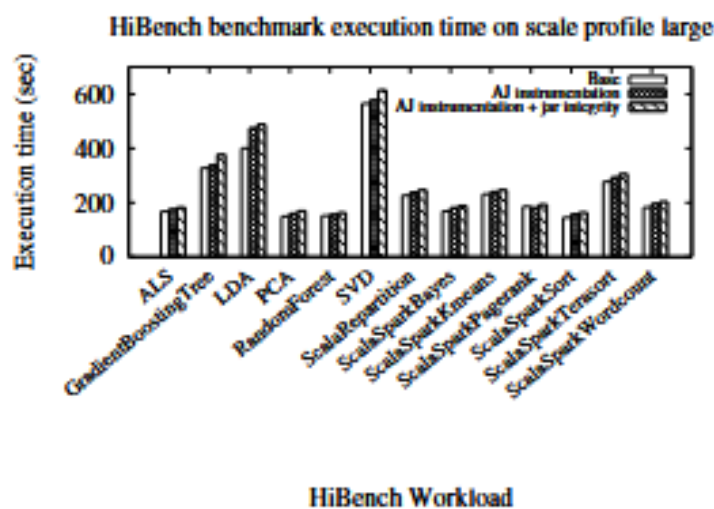
SecureDL



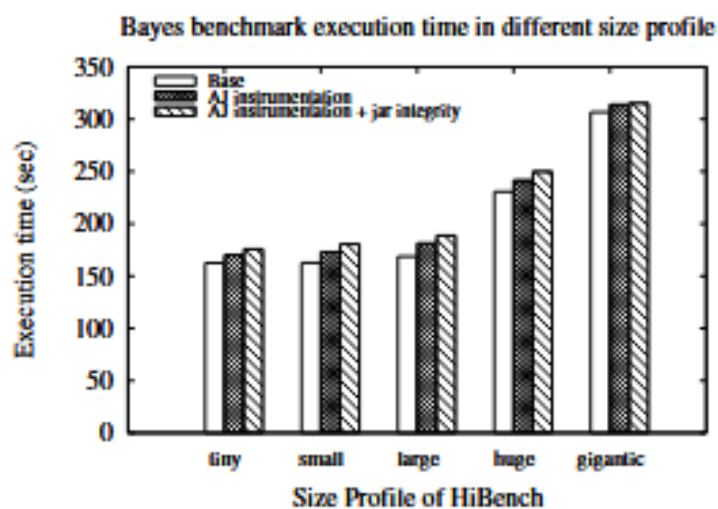
Evaluation

- Comparing the performance overhead based on:
 - Different job workloads:
 - Varying tasks and input size, provided by [HiBench](#)
 - Size of the dataset provided:
 - From 10 to 50 GB (using [TPCH queries](#) + [masks](#))
 - Size of clusters:
 - From 3 to 7 nodes (same as before)

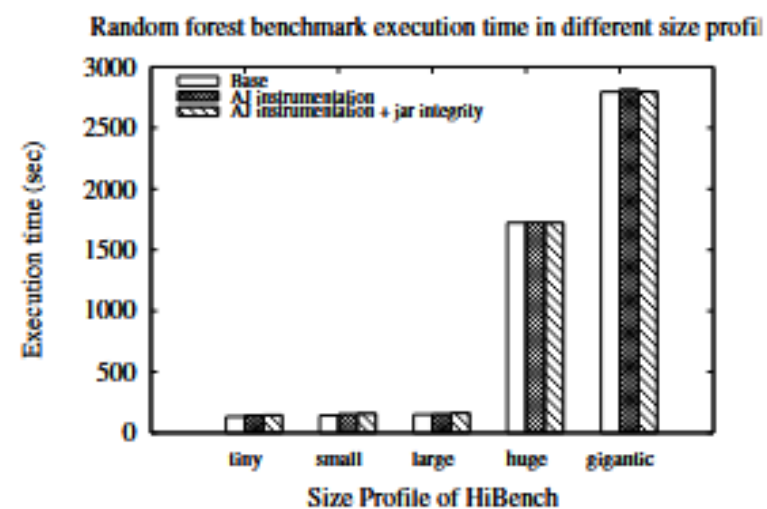
Overhead: workloads



(a) HiBench Large Profile



(b) Bayes on different scale

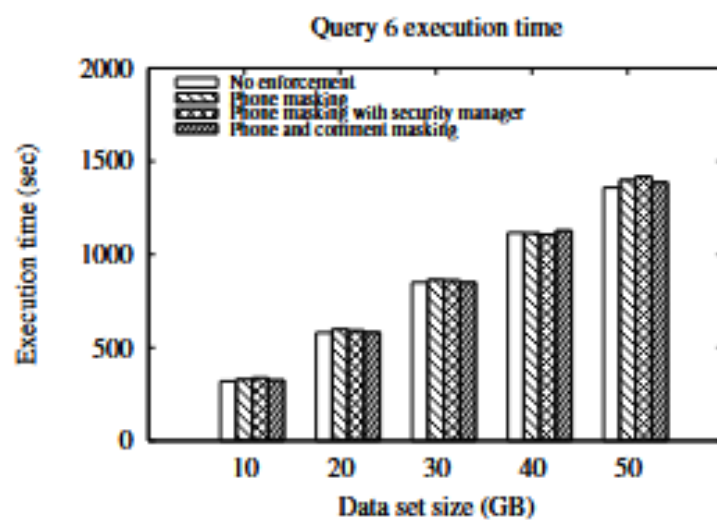


(c) RandomForest tree on different scale

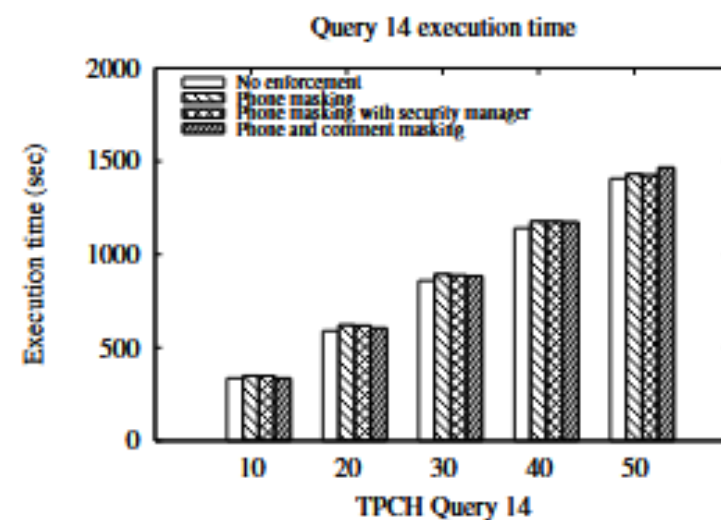
Overhead: input size



(a) Query 2

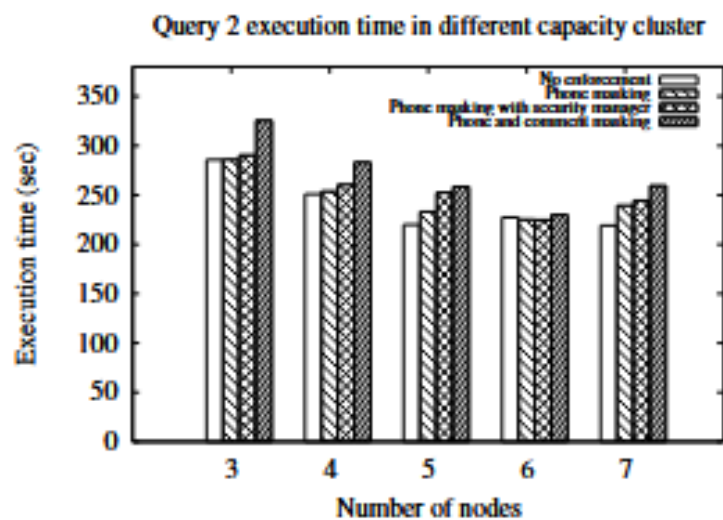


(b) Query 6

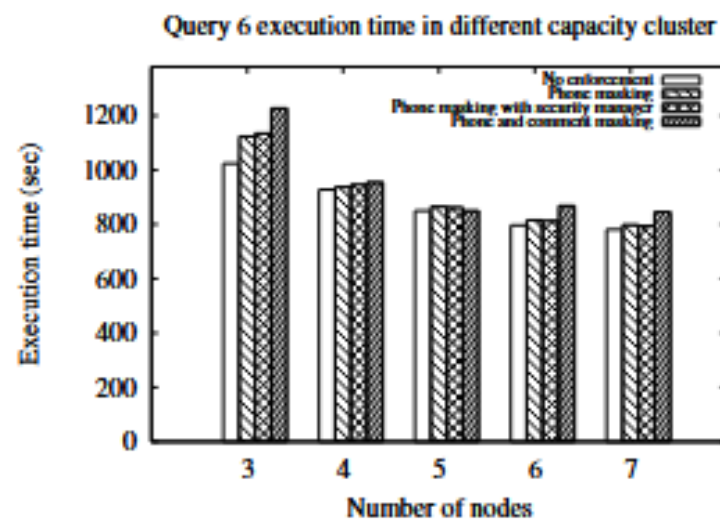


(c) Query 14

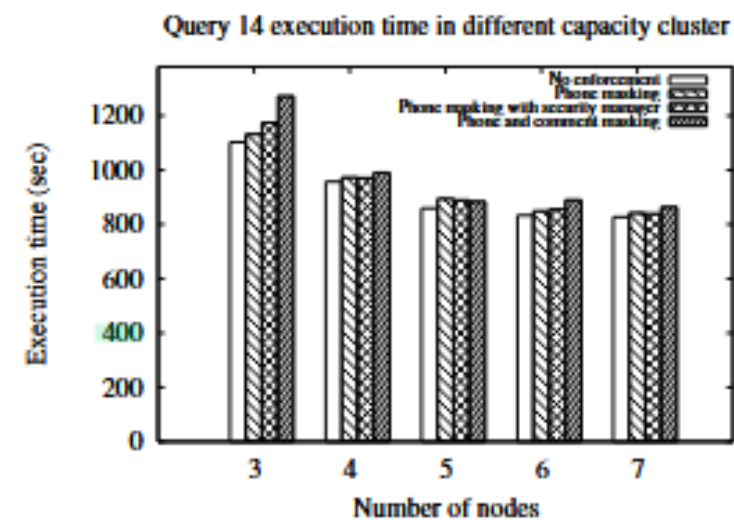
Overhead: cluster size



(a) Query 2



(b) Query 6



(c) Query 14

Thanks for the attention!



Query masks

```
email:
  name: EmailMask
  type: regex_mask
  data_type: email
  detection_regex: "\\b [^\s] \
+@[a-zA-Z0-9] \
[a-zA-Z0-9- ]{0,61} \
[a-zA-Z0-9]{0,1} \
\\.([a-zA-Z ]{1,6} | \
[a-zA-Z0-9- ]{1,30} \
\\. [a-zA-Z]{2,3}) \b"
  replacement_pattern: '*@*c'
```

```
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: ShowLast40f12Digits
  visible_anchor: end
  visible_chars: 4
```


TPCH queries

```
SELECT S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY, P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT
FROM ...
WHERE P_SIZE = :size
      AND P_TYPE LIKE :type
      AND R_NAME = :region
      AND PS_SUPPLYCOST = (
        SELECT MIN(PS_SUPPLYCOST)
        FROM ...
        WHERE ...
      )
ORDER BY ...
```

```
SELECT SUM(L_EXTENDEDPRI * L_DISCOUNT) AS revenue
FROM LINEITEM
WHERE L_SHIPDATE >= DATE '1994-01-01'
      AND L_SHIPDATE < DATE '1995-01-01'
      AND L_DISCOUNT BETWEEN 0.05 AND 0.07
      AND L_QUANTITY < 24;
```

```
SELECT 100.00 * SUM(CASE
                    WHEN P_TYPE LIKE 'PROMO%' THEN L_EXTENDEDPRI * (1 - L_DISCOUNT)
                    ELSE 0
                    END) / SUM(L_EXTENDEDPRI * (1 - L_DISCOUNT)) AS promo_revenue
FROM LINEITEM
JOIN PART ON L_PARTKEY = P_PARTKEY
WHERE L_SHIPDATE >= DATE '1995-09-01'
      AND L_SHIPDATE < DATE '1995-10-01';
```