Secure Data Analytics in Apache Spark with Fine-grained Policy Enforcement and Isolated Execution

Byeongwook Kim[†]
Seoul National University
rlaquddnr904@snu.ac.kr

Jaewon Hur[†]
Seoul National University
hurjaewon@snu.ac.kr

Adil Ahmad
Arizona State University
adil.ahmad@asu.edu

Byoungyoung Lee[‡]
Seoul National University
byoungyoung@snu.ac.kr

***Abstract*—Cloud based Spark platform is a tempting approach for sharing data, as it allows data users to easily analyze the data while the owners to efficiently share the large volume of data. However, the absence of a robust policy enforcement mechanism on Spark hinders the data owners from sharing their data due to the risk of private data breach. In this respect, we found that malicious data users and cloud managers can easily leak the data by constructing a policy violating physical plan, compromising the Spark libraries, or even compromising the Spark cluster itself. Nonetheless, current approaches fail to securely and generally enforce the policies on Spark, as they do not check the policies on physical plan level, and they do not protect the integrity of data analysis pipeline.**

This paper presents LAPUTA[1], a secure policy enforcement framework on Spark. Specifically, LAPUTA designs a pattern matching based policy checking on the physical plans, which is generally applicable to Spark applications with more fine-grained policies. Then, LAPUTA compartmentalizes Spark applications based on confidential computing, by which the entire data analysis pipeline is protected from the malicious data users and cloud managers. Meanwhile, LAPUTA preserves the usability as the data users can run their Spark applications on LAPUTA with minimal modification. We implemented LAPUTA, and evaluated its security and performance aspects on TPC-H, Big Data benchmarks, and real world applications using ML models. The evaluation results demonstrated that LAPUTA correctly blocks malicious Spark applications while imposing moderate performance overheads.

I. Introduction

Cloud based big data analytics [62] is an emerging approach to share large volume of data, as the data users can easily analyze the data while the owners can efficiently share the data on the cloud. In particular, Spark [65] is one of the most popular big data analytics platform that is well suited for these use cases. For example, several research agencies from pharmaceutical companies utilize Spark to analyze medical datasets shared from hospitals [31]. As another example, it is a common approach for the banks to federate their databases into a shared Spark platform, efficiently analyzing large volume of data [3]. There are already numerous cloud managers who provide a full-featured Spark platform as a service [22], [39].

However, sharing the data without proper policy enforcement may incur significant financial costs to the data owners as malicious data users can breach the private (or proprietary) information. These security issues are particularly important considering the fact that datasets (e.g., medical data [21], [31], and financial data [1]) often contain private information regulated under GDPR [45], CCPA [25], and HIPAA [7]. If not observed, the data owners can be faced with serious charges, which has been showcased in the recent Cambridge Analytica lawsuit of Meta, costing 725 million dollars for settlement [16]. Thus, the inability to enforce the policies is the major obstacle for the data owners from sharing their data.

In this respect, we found that current architecture of Spark [65] is not sufficient to enforce the policies on data. Especially, the problem lies in the monolithic architecture of Spark, which gives full control to the data users to manipulate entire data analysis pipeline. Thus, the users can easily retrieve a policy violating analysis result by i) building a malicious query plan in Spark applications (using Spark libraries [65], [10] as usual), ii) directly compromising the Spark libraries (e.g., Spark SQL [10], MLlib [38]), or even iii) manipulating worker nodes that actually perform computations on the data.

To this end, we found that it is essential to protect the entire data analysis pipeline, which should meet the following two requirements: i) checking the query plans against the policies, and ii) ensuring the validated plans are correctly executed on the data. However, existing works to enforce the policies in database (in general) have failed to fulfill these requirements [37], [52]. In particular, most of the works are limited to enforce the policies only on SQL context [37], not on the query plans generally constructed in Spark applications [65], [10]. Furthermore, they do not provide general policy definitions, limiting their uses for specific cases [37], [52]. Worse yet, none of the works have considered protecting the data analysis pipeline from a malicious Spark application itself, as it was commonly assumed benign [60], [61]. However, in our threat model, it is not true as the malicious data user has full control

[†] These authors contributed equally to this work.

[‡] Corresponding author

[1] We named it after the famous Japanese animation, ”Laputa: Castle in the Sky”

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242496
www.ndss-symposium.org

over the Spark applications as well as the worker nodes.

In this paper, we design LAPUTA, a secure policy enforcement framework on Spark. LAPUTA can be characterized by two correlated design decisions—i.e., integrating policy enforcement logic and confidential computing based compartmentalization to meet the aforementioned requirements. First, LAPUTA designs a new pattern matching [64] based approach to define and enforce the policies on Spark physical plans. Especially, LAPUTA employs regular expression in defining the policies so that the data owners can express fine-grained policies that work on general Spark applications. Second, LAPUTA leverages compartmentalization with confidential computing [5], [29], [35], [9] to protect the data analysis pipeline from malicious data users. Specifically, we compartmentalize the Spark applications into two address spaces such that the untrusted user cannot compromise the core LAPUTA logic, which includes policy checking and query execution. In addition, the safe address space (running core LAPUTA logic) is further protected by confidential computing (e.g., AMD SEV-SNP [5], [4]), thereby ensuring its integrity from the attackers controlling entire software stacks.

We implemented LAPUTA into two extensible modules of Spark, where each is for the data user and the data owner respectively. For the user's side, LAPUTA preserves the compatibility so that the users do not need to modify their code (using Spark libraries [65], [10]), but only have to import the corresponding LAPUTA module. On the other hand, for the owner's side, LAPUTA protects the entire data analysis pipeline (including the data itself) with AMD SEV-SNP [4], which includes query processing, policy checking, and data processing in the distributed nodes.

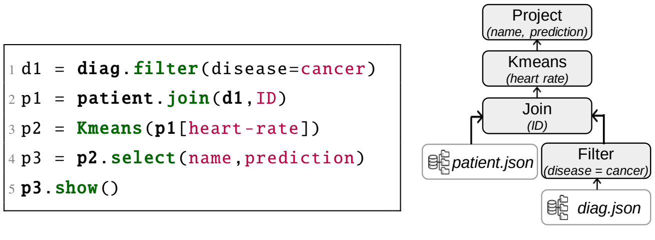
In order to evaluate the practical impact of LAPUTA, we performed security and performance evaluations on the prototype implementation. As a security evaluation, we checked whether LAPUTA correctly enforces 7 synthetic policies on TPC-H benchmark [51], which consists of 8 tables and 22 complex queries. The security evaluation clearly demonstrated that LAPUTA can block malicious data retrievals by blocking 9 queries from the benchmark. For the performance evaluation, we measured the overheads of LAPUTA on i) TPC-H benchmark [51], ii) Big Data benchmark [6], and iii) real world applications using recommendation [27], [34], and clustering models [41]. The evaluation showed that LAPUTA imposes 35% of latency and 25% of throughput overhead on average, demonstrating its strong feasibility.

II. BACKGROUND

In this section, we provide brief backgrounds on Spark (§II-A), and confidential computing (§II-B).

A. Distributed Big Data Analysis with Spark

Apache Spark [65] is one of the most successful and efficient distributed big data analytics platform. The platform provides an abstracted programming model for the users so that they can focus on analyzing the data while the Spark engine handles the low level operations of managing distributed nodes. To be specific, Spark requires the users to construct a *physical plan*, which is a directed acyclic graph (DAG) specifying how the data should be computed to get the output. Then, Spark automatically splits the physical plan into a number of small tasks, and schedules them to be executed in the distributed worker nodes.


```
1 d1 = diag.filter(disease=cancer)
2 p1 = patient.join(d1,ID)
3 p2 = Kmeans(p1[heart-rate])
4 p3 = p2.select(name,prediction)
5 p3.show()
```

(a) Spark application

(b) Physical plan

Figure 1: Example of a Spark application and the constructed physical plan.

An important design decision of Spark is that it provides libraries [10], [38], [65] for the users to easily construct the physical plans. Thus, the users develop their own Spark applications using the Spark libraries, and the libraries automatically construct the physical plans and run the tasks on the nodes. By providing Spark libraries, it offers high flexibility for the users to employ arbitrary code and (other) libraries [10], [43], [36] in building their Spark applications, which facilitates the data analysis process. For example, it is a common pattern to use `PySpark` [65], [10] with graphical libraries [8] to visualize the analysis results. Furthermore, the extensions of the Spark libraries (e.g., Spark SQL [10], and MLlib [38]) empower the users to conduct more thorough data analysis by supporting various interfaces.

While Spark applications can be developed using several libraries [33], [46], [44], [32], all of them internally perform aforementioned two operations: i) physical plan construction, and ii) task generation and execution. As the first operation, Spark application constructs a physical plan (i.e., DAG) whose nodes denoting the computations to be perfomed on the data and the edges denoting the data flow (as shown in Figure 1-(b)). In particular, Spark libraries provide two sorts of APIs: `transform` [65] and `action` [65]. The APIs are freely invoked by the user as shown in Figure 1-(a). Upon the invocations of `transform` APIs (e.g., `filter` in line 1 of Figure 1-(a)), the libraries continuously construct the physical plan by inserting a node into it (i.e., `Filter` node inserted in Figure 1-(b)). Then, when an `action` API is invoked (e.g., `show` in line 5 of Figure 1-(a)), the plan is optimized and filled with the information for actual execution (i.e., complete plan in Figure 1-(b)), and the Spark application proceeds to the next operation.

In subsequent operation, Spark application splits the physical plan into multiple tasks and executes them in the distributed worker nodes. Especially, each task is a byte code [17], [65], [10] which runs on an interpreter (e.g., Java Virtual Machine [57]), containing the information about how the

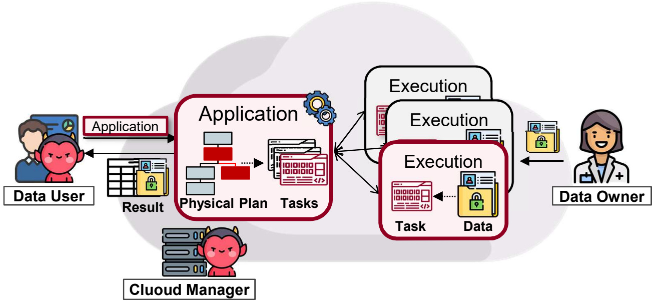


Figure 2: System model of LAPUTA.

data should be processed locally (e.g., the source dataset, the operations, and the destination node that should receive the output). Thus, the tasks are concurrently executed in the distributed nodes, and the final result (i.e., final output of the computations following the nodes in the physical plan) is returned after all the tasks are completed.

B. Confidential Computing

Confidential computing is a technique to protect the data in cloud using the security features provided by hardware (e.g., Intel SGX [35], TDX [29], AMD SEV [5], and NVidia H100 [42]). Especially, the confidential computing enabled hardware constructs an isolated execution environment, called enclave, which is protected against the privileged components such as operating systems and hypervisors. Thus, it ensures the confidentiality of enclave owner's program and data even in a compromised environment. Furthermore, the enclave provides a remote attestation, by which its owner verifies the integrity of the loaded program, and sends their secret data into it [35], [5], [29]. Thanks to its strong security benefits, emerging cloud applications are employing (or expected to employ) confidential computing to protect the customer's data (e.g., Apache Hadoop [48], Machine Learning [38], etc).

III. MOTIVATION

This paper focuses on the security problem of enforcing data policies while using Spark for data analysis. In this section, we provide the system model and problem setting of LAPUTA (§III-A). Then, we discuss the possible attack vectors when enforcing policies on Spark (§III-B), and briefly introduce our approaches (§III-C).

A. System Model

We consider the scenario of sharing a dataset on Spark [65], [10], where the owner and the user of the data are different. Especially, we consider three parties involved in this scenario as shown in Figure 2: i) data owner, ii) data user, and iii) cloud manager.

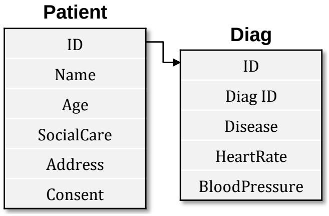
- **Data Owners** have a valuable dataset, which they want to share for data analytics by third-party data users. Meanwhile, they want specific policies are enforced on data retrieval, as the dataset contains private and proprietary information. This requirement may stem from internal considerations, as well as from privacy laws such as GDPR [45], HIPAA [7], and CCPA [25]. Common real world examples of such data owners are hospitals [31] that want to share medical dataset for research, but also want to protect the private information of their patients.
- **Data Users** analyze the owner's data using Spark. They develop Spark applications using various Spark libraries (e.g., Spark SQL [10], MLlib [38]), and the applications may contain arbitrary code and libraries to facilitate the data analysis (e.g., libraries for graphical interfaces [8]). Common real world examples of data users are pharmaceutical research agencies [31], [23] that often analyze the medical dataset from the hospitals.
- **Cloud Manager** runs the Spark engine [65] on behalf of the data owner. This is because the conventional data owners (such as hospitals [40] and banks [50]) do not have sufficient computing infrastructure. Thus, it is a common approach to outsource the database [12], [30], and there are already numerous cloud managers who provide data outsourcing services based on Spark (e.g., Databricks [22], and Azure Synapse [39]).

Problem Setting. In this system model, the data owners are concerned that the data users and cloud managers might obtain policy violating information from the data. Since the current Spark platform lacks sufficient policy enforcement (except for coarse-grained access controls on the data [60]), malicious data users would easily obtain policy violating analysis results from the data by implementing malicious Spark applications. Furthermore, malicious cloud managers may also breach the data as they control the entire software and hardware stacks.

B. Secure Policy Enforcement on Spark

To this end, we focus on designing a secure policy enforcement mechanism on Spark. Thus, we first discuss the possible attack vectors raised by malicious data users and cloud managers using an example scenario of sharing medical dataset. Then, we discuss the limitations of previous works for enforcing the policies on Spark.

Example Scenario of Sharing Medical Dataset. Suppose a hospital owns a medical dataset, whose schema is shown in Figure 3-(a), and they want to share the data with third-party pharmaceutical research agency. Especially, the research agency is finding suitable patients for their targeted clinical trials [31], [21] as a drug works differently on each patient [40], [31], [12]. Thus, the agency has to retrieve the name and address of the patients, who are the most appropriate to the investigational drug based on the medical information. However, the hospital also wants to enforce the policies as shown in Figure 3-(b) as the data contains private information of the patients. Specifically, the policies prevent the research agency from abusing (e.g., illegally using the data in machine learning) or leaking the patient's private data (e.g., exposing the disease of the patients). Thus, they are hesitating to share the data as it is not sufficient for current Spark platform to enforce the policies.

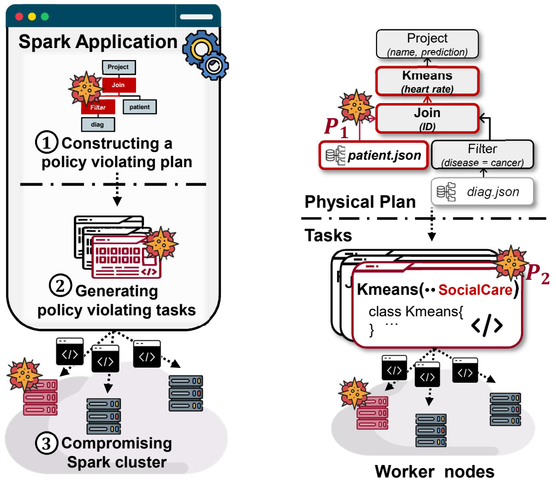


(a) DB schema of the medical dataset.

	Example policies
$\mathbf{P_1}$	Only the records of the patients who have consented can be used in machine learning.
$\mathbf{P_2}$	Patient's social care status and name must not be used in machine learning.
$\mathbf{P_3}$	When the `patient` and `diag` tables are joined, the patients' disease must not be revealed.

(b) Policies that the hospital wants to enforce.

Figure 3: Example scenario of sharing medical dataset.



(a) Attacks in data analysis pipeline.

(b) Example attacks breaking policies 1 and 2.

Figure 4: Attacks to break the policies on Spark.

Possible Attacks to Break Policies on Spark. Without a secure policy enforcement, a malicious research agency (and cloud manager) would easily break the policies (and leak the data) as shown in Figure 4. To be specific, they would bring the analysis results not satisfying the policies by i) building a malicious Spark application that constructs a policy violating physical plan (i.e., ①), or ii) generates policy violating tasks (even from a benign physical plan, ②), or even iii) compromising the Spark cluster itself, manipulating the executions in the worker nodes (i.e., ③).

Spark applications may construct a physical plan that does not satisfy the policies (i.e., ① in Figure 4-(a)). As a straightforward example, as shown in the first attack of Figure 4-(b), a malicious application would construct a physical plan that does not check the consents from the patients before using their records in machine learning (i.e., `Kmeans`), breaking the policy $\mathbf{P_1}$—i.e., the `patient` table should have been filtered on the `consent` field.

However, checking only the policy compliance of the physical plan does not ensure the safety of the final results, as a malicious Spark application can tamper with the actual tasks (e.g., attack ② in Figure 4-(a)). For instance, Spark libraries may be compromised to modify the tasks, as shown in the second attack of Figure 4-(b), to use the patient's social care status in machine learning (i.e., `Kmeans`) even if the original plan was not, breaking the policy $\mathbf{P_2}$. Furthermore, even if the tasks are benign, actual execution of the tasks in worker nodes can be compromised (i.e., ③ in Figure 4-(a)) since the malicious cloud manager can deploy compromised images to the worker nodes in Spark cluster.

Limitations of Previous Works. Thus, in order to enforce the policies on Spark, LAPUTA must ensure the policies are correctly obeyed throughout the entire data analysis pipeline. For that, following two requirements must be guaranteed: i) the physical plan constructed in the Spark application must be checked against the policy compliance, and ii) the entire procedure from the policy checking to the task generation and execution must not be tampered with. Hence, LAPUTA can ensure the policy compliant physical plan is correctly executed on the data.

However, none of the previous works have designed a policy enforcement mechanism considering these two requirements. First of all, existing works on the policy enforcement mechanism [52], [37], [60], [61] do not provide a method to define the policies on physical plan. While most of them provide the policy enforcement mechanism which works on SQL syntax (e.g., SQL query rewriting [37]), they did not consider general policy enforcement on the physical plans. Although LAPUTA can employ these works by restricting the data users to use only the SQL, it would severely harm the utility of the data as the other Spark functionalities are widely used for data analytics also—e.g., Spark SQL occupies only 18% of total usage in Notebooks [18].

Furthermore, none of them have deliberated on how to enforce the policies generally on the physical plans. For example, the physical plan given in Figure 4-(b) was actually breaking the policy $\mathbf{P_3}$, exposing the disease information of the patients—i.e., the `diag` table is filtered on the `disease` before being joined to the `patient` table, and the patient's `name` is projected, printing who has been diagnosed with `cancer`. In order to prevent these cases, the policies should be expressible enough to be matched over the entire physical plan, as there are countless variants of the plans and data owner's requirements. However, the existing works have only provided fixed and adhoc policies such as merely preventing specific tables from being joined [37], [60], [61].

Second, none of the works have protected the data analysis pipeline against the Spark users themselves, assuming the Spark applications and worker nodes are trusted. While there were several works that focused on SQL checking (assuming the user is malicious), they assumed only the SQL query can be malicious and the query processing is safely executed [60], [52], [37], [61]. In contrary, we assume the entire procedure of query processing can be tampered with as the adversaries have full control over the entire Spark pipeline (from the Spark applications to the worker nodes). While Opaque [66] proposed to use confidential computing to protect the query processing from malicious cloud managers, they did not care about the integrity of task generation from the physical plan as they assumed the Spark application is benign. Our threat model is fundamentally different from Opaque in that we assume the user, who develops the Spark applications themselves, can be malicious.

C. Laputa*'s Approaches*

In order to achieve the requirements stated in §III-B, we design following two approaches. First, Laputa leverages a pattern matching-based policy checking on physical plans, which employs regular expression [64] to define data usage policies. Specifically, Laputa checks the policies by examining whether the nodes in the physical plan match the given pattern of symbols, defined using regular expressions. The rationale behind defining policies using regular expressions is to offer better flexibility to data owners, allowing them to express any policy expressible within regular languages. For example, a pattern to match the plan in Figure 4-(b), which breaks the policy $\mathbf{P_3}$, can be defined as $*s_3*s_1*$—it first identifies the `Filter` (i.e., s_3) and then the `Join` (i.e., s_1). We explain the details of regular expression based policy definition and Laputa's policy check logic in §IV-A.

Second, Laputa compartmentalizes the Spark applications to protect the integrity of data analysis pipeline, which includes the policy check logic, task generation, and executions in worker nodes. In order to guarantee the integrity of data analysis pipeline, it is essential to sandbox [28], [63] the Spark application as it runs the (potentially malicious) data user's code and the Spark libraries in the same address space. In other words, without sandboxing, a malicious code would easily tamper with the Spark libraries to bypass the security critical logics (e.g., policy checking) and directly control the worker nodes to run policy violating tasks. Thus, Laputa sandboxes the user's code by compartmentalizing the Spark application into two different address spaces (e.g., different processes running in different VMs), preventing the code from directly accessing security critical logics. In addition, Laputa further protects the security critical logics using confidential computing to prevent cloud managers from illegally affecting the Laputa's logic and worker nodes.

Laputa's compartmentalization based sandbox provides high flexibility for the data users as they can run their original Spark applications with minimal modification. To be specific, Laputa modifies the Spark libraries so that the security critical logics (e.g., policy checking) can be seamlessly executed in the different address space, protected by the confidential computing. Our evaluation demonstrates that the data users can implement the Spark applications on Laputa without intrusively modifying their code. We explain the design details of compartmentalization based sandbox in §IV-B.

IV. Design of Laputa

Laputa designs a new Spark framework, which securely enforces the data owner's policies while the users analyze the data. To be specific, Laputa works as the conventional Spark platform on cloud, where the data owner pre-uploads the data and the users analyze it by running their own Spark applications implemented using either `Scala` [43], `Python` [55] or Spark-SQL APIs [10]. However, the end-to-end flow of the data is protected using Laputa (and confidential computing) such that the data users (and cloud managers) can obtain only the analysis results satisfying owner defined policies.

Laputa's design can be characterized by the following two aspects: i) physical plan checking with LaputaChecker (§IV-A), and ii) confidential computing based compartmentalization with LaputaExecutor (§IV-B). In the following, we briefly discuss the threat model, and explain these two components.

Threat Model. Laputa employs enclaves [35], [29], [5] in the untrusted cloud environment to ensure its confidentiality and integrity to the data owners. Thus, it does not trust any other entities outside the enclave, which includes OSes and hypervisors of the cloud managers, and the Spark applications implemented by the data users. On the other hand, general security limitations of confidential computing, such as microarchitectural side channels [59], [56], [54], and Iago attacks [15] are out-of-scope. Denial-of-Service attacks [47] are also out-of-scope.

We assume the implementation of Laputa's logic is bug-free such that the data users cannot compromise its internal operations using vulnerabilities. While Laputa enables enforcing the policies on Spark, data owners are in charge of defining the policies to correctly reflect their requirements. Laputa does not ensure security against improper policy definition. On the other hand, adversaries may construct a covert channel to leak side information of the data (e.g., size of the table). Laputa currently does not protect against such covert channels and we discuss the potential mitigations in §VIII.

A. *Physical Plan Checking with* LaputaChecker

LaputaChecker checks all physical plans constructed in the Spark applications against the owner defined policies. In particular, data owners provide the allowed (or disallowd) patterns of the nodes in the physical plans, using regular expression, and LaputaChecker checks whether the nodes in the constructed physical plans match those patterns. To this end, we elaborate how the data owners define the policies for LaputaChecker, and how the physical plans are checked against the policies.

Policy Definition for LaputaChecker. In order to express various policies needed by the data owners, LaputaChecker uses the pattern of the nodes as the policy, which is based

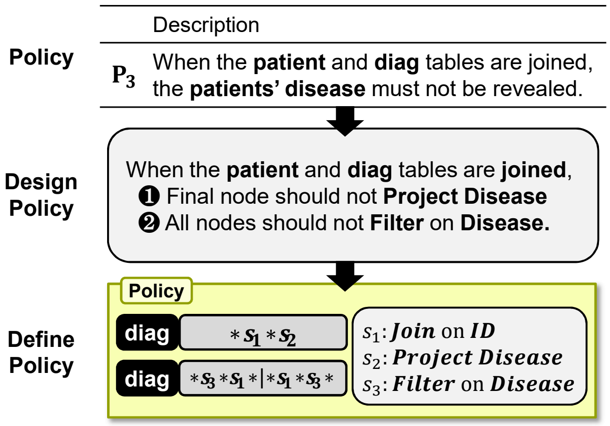

Description
Policy
P_3
When the **patient** and **diag** tables are joined, the **patients' disease** must not be revealed.
Design Policy
When the **patient** and **diag** tables are **joined**,
❶ Final node should not **Project Disease**
❷ All nodes should not **Filter** on **Disease.**
Define Policy
Policy
diag
$*s_1*s_2$
diag
$*s_3*s_1*|*s_1*s_3*$
s_1: *Join* on *ID*
s_2: *Project Disease*
s_3: *Filter* on *Disease*

Figure 5: The procedure of defining policies.

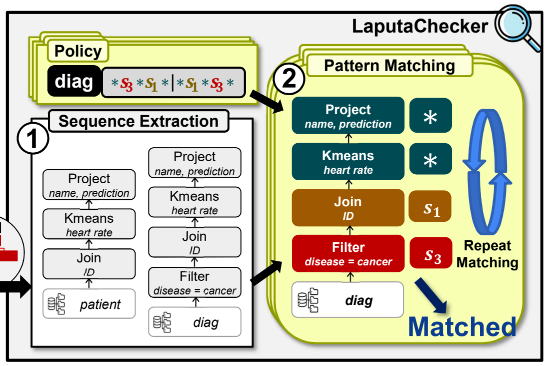


Figure 6: Design of LAPUTACHECKER.

on the regular expression [64]. Suppose $\mathbf{P_3}$ in Figure 5 (i.e., explained in §III-B), which mandates the patient's disease must not be revealed when the two tables (i.e., `patient` and `diag` tables) are joined. For this policy, the owner has to enforce that, if the tables are joined, i) the `disease` field must not be retrieved as output (as it can be linked to the private information of the patients), and ii) the `disease` field must not be used for filtering (as it can reveal the patients who have the specific disease).

To this end, the owner has to express the requirements into the patterns that can match on the physical plan. Especially, the pattern for the first requirement would denote that the final node (i.e., root node, which is processed the last) in the plan must not `Project` the `disease` field, if the tables are joined (i.e., ❶ in Figure 5). The pattern for the second requirement would denote that all the nodes in the plan must not `Filter` on the `disease` field, if the tables are joined (i.e., ❷). As such, in order to express the requirements into the patterns, LAPUTACHECKER defines the policy as follows:

- Policy $P = \langle t, r \rangle$ is a tuple, where t denotes a table (i.e., leaf node in the plan) whose following nodes are matched against the pattern, and r denotes a pattern expressed in regular expressions.

Given the policy $\langle \mathbf{t}, \mathbf{r} \rangle$, LAPUTACHECKER checks whether all the nodes that process the table $\mathbf{t}$ (i.e., sequence of the nodes from $\mathbf{t}$ to the root node) match the pattern $\mathbf{r}$ (i.e., regular expression of symbols). In this respect, the first policy would be $\langle \mathtt{diag}, *\mathtt{s_1}*\mathtt{s_2} \rangle$, where the symbol $\mathtt{s_1}$ represents the `Join` of two tables, and $\mathtt{s_2}$ represents the `Project` of `disease` field—i.e., join the tables, and then project `disease` the last. The second policy would be $\langle \mathtt{diag}, *\mathtt{s_1}*\mathtt{s_3}* \mid *\mathtt{s_3}*\mathtt{s_1}* \rangle$, where $\mathtt{s_1}$ represents the `Join` as before, and $\mathtt{s_3}$ represents the `Filter` on `disease` field—i.e., filtering on `disease` before or after joining the tables. Thus, LAPUTA can forbid processing the physical plans matching these patterns.

Pattern Definition for LAPUTACHECKER. Based on the policies provided, LAPUTACHECKER performs pattern matching on the physical plan as shown in Figure 6. Especially, LAPUTACHECKER splits the plan into the node sequences depending on the leaf tables (i.e., ① in Figure 6), and matches the sequences against the patterns of the policies which have the same table (i.e., ②). Taking the aforementioned physical plan in Figure 4 as an example, the node sequence of `diag` table would be the right one in `Sequence Extraction` of Figure 6. Then, LAPUTACHECKER would match the sequence against the patterns (e.g., $*\mathtt{s_3}*\mathtt{s_1}* \mid *\mathtt{s_1}*\mathtt{s_3}*$) as explained earlier. In particular, LAPUTACHECKER matches each node in the sequence (e.g., `Filter(disease=cancer)` node, `Join(ID)` node, and others) against each symbol in the pattern (e.g., $\mathtt{s_3}$, $\mathtt{s_1}$, and $*$).

For that, LAPUTA formally defines the symbol, and the matching condition between the symbol (in the pattern) and the node (in the sequence). LAPUTA defines the symbol using an operator, available fields, and optional expression as follows:

- Symbol $s = \langle o, [\{f\}] \rangle : (e)$, where o denotes the operator, $[\{f\}]$ denotes the available fields, and the optional argument e denotes the expression that should be applied on the fields. If e is not provided, any expression is allowed.

Given this, the matching condition of the symbol and the node is as follows:

- A node n matches a symbol s, if
  i. the operator of n is the same as the operator of s.
  ii. the fields used in n belong to the available fields of s.
  iii. if s has an optional expression, the expression applied to the fields in n is the same as the optional expression of s.

Thus, a single symbol can be matched to multiple nodes that satisfy the matching conditions. Taking the policies in Figure 5 as example, the symbol $\mathtt{s_1}$ can be defined as $\mathtt{s_1} = \langle \mathtt{Join}, [\{\mathtt{ID}\}] \rangle$ that matches all nodes which join the tables using `ID` field as a key. The symbol $\mathtt{s_3}$ would be defined as $\mathtt{s_3} = \langle \mathtt{Filter}, [\{\mathtt{disease}\}, \mathtt{ALL_f}] \rangle$ that matches all nodes which use `disease` field in filtering—i.e., we use $\mathtt{ALL_f}$ as a wildcard field which can be any multiple fields except the already used one. Thus, the pattern $*\mathtt{s_3}*\mathtt{s_1}*$ would match any sequences of nodes that contain the node, which `Filter` on `disease`, followed by the node, which `Join` on `ID`, as shown in `Pattern Matching` of Figure 6.

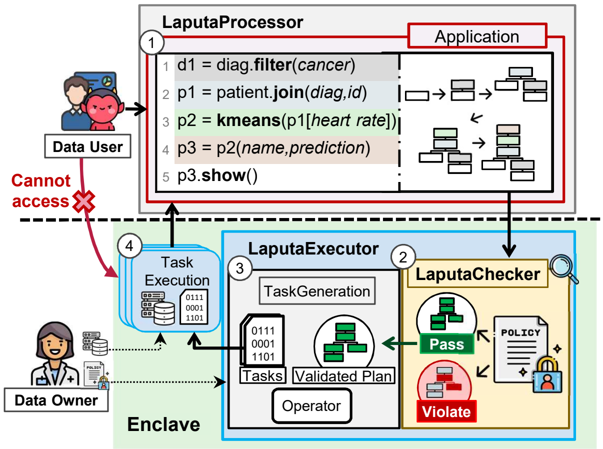
In addition, LAPUTA provides the optional expression (i.e., `e` from the definition of symbol), and wildcards (i.e., $\mathtt{ALL_{op}}$

Table I: A usage of optional expressions. The list of entire operators is shown in Table IX.

Operator	Example	Description
Filter	⟨Filter, [{age}]⟩ ⟨Filter, [{age}]⟩ : (age>=20)	Filter using age with any filter condition. Filter only with filter condition age>=20.
Join	⟨Join, [{ID}]⟩	Join must only use identity expression. Optional expression is not allowed on Join.
Others	⟨Project, [{age}]⟩ : mask(age)	Project using age with mask function.

Table II: A usage of $\mathbf{ALL}_{\mathtt{f}}$. Details of available fields and $\mathbf{ALL}_{\mathtt{f}}$ are explained in §A.

Available fields of symbol	Set of fields used in nodes that match
$\mathtt{ALL_f}$	Any set of fields
[{name, age}]	{name}, {age}, {name, age}
[{name, age}, $\mathtt{ALL_f}$]	Any set of fields that contains either name or age



```
1 d1 = diag.filter(cancer)
2 p1 = patient.join(diag,id)
3 p2 = kmeans(p1[heart rate])
4 p3 = p2(name,prediction)
5 p3.show()
```

Figure 7: Compartmentalization design of LAPUTA.

denoting any operator, and $\mathtt{ALL_f}$ denoting any fields) in defining the symbols, which facilitates making the patterns. The optional expression can be used to specify a node that uses only a given expression as shown in Table I. For instance, if the data owners want to enforce the users to filter out teenage patients' records, they can use a symbol ⟨Filter, {age}⟩ : (age>=20) in their patterns, and only the physical plans that contain a node filtering on age>=20 will be matched.

The wildcards are used to match any nodes that use a given field (e.g., ⟨$\mathtt{ALL_{op}}$, disease⟩ denoting any nodes using disease field), or any nodes that use a given operator (e.g., ⟨Filter, $\mathtt{ALL_f}$⟩ denoting any Filter nodes). Especially, when $\mathtt{ALL_f}$ is used in conjunction with other fields, any subset that contains any one of the used field matches (i.e., illustrated in Table II). Thus, data owners can define various policies using these features, which is evaluated in §VI-C. We refer to §A for the formal description of LAPUTA's policy language.

Checking Physical Plans. Based on the results of pattern matchings, LAPUTACHECKER determines whether a given physical plan satisfies all the policies or not. In particular, data owners provide whether a policy i) allows only the plans matching the given pattern, or ii) disllows the plans matching the given pattern. Thus, LAPUTACHECKER validates a phyiscal plan only when it satisfies all the policies with its side accordingly (i.e., allow or disallow). Taking the policy example of Figure 5, both policies disallow the plans matching the patterns, and a plan which matches any of the pattern would be rejected.

B. *Compartmentalization with* LAPUTAEXECUTOR

In order to enforce the data policies throughout entire data analysis pipeline, LAPUTA should ensure the physical plans are checked (by LAPUTACHECKER as explained in §IV-A), and the tasks are correctly generated and executed from the validated plans. To this end, we compartmentalize the Spark applications, because in current Spark architecture, a malicious code can easily tamper with the security critical logics (e.g., Spark libraries) running in the same address space. More specifically, our compartmentalization splits the Spark application into two components (as shown in Figure 7), running in different address spaces: i) LAPUTAPROCESSOR, which runs the potentially malicious user provided code, and ii) LAPUTAEXECUTOR and worker nodes, which safely perform policy checking on the physical plans, and tasks generation and execution (through distributed nodes). In addition, LAPUTAEXECUTOR and worker nodes run in the enclaves to protect its integrity against the attackers controlling entire software stacks.

LAPUTAPROCESSOR: Running Untrusted User's Code. Untrusted user's code always runs in LAPUTAPROCESSOR (i.e., ① in Figure 7), which is separated from the data as well as the data analysis pipeline of the physcal plans. Thus, the user's code are strictly forbidden from retrieving arbitrary data or tampering with the data analysis pipeline. On the other hand, the users can implement arbitrary data analysis code as well as other functionalities (e.g., visualizing analysis results [8]), since LAPUTAPROCESSOR does not restrict how the application is implemented. At the user's point of view, LAPUTA's libraries in LAPUTAPROCESSOR provide the same programming interfaces as Spark libraries [10], [38], with which the users can analyze the data as usual.

The only difference is that the LAPUTA's libraries internally relays the constructed physical plan to the LAPUTAEXECUTOR upon the request to execute the plan (i.e., invocation of action APIs [65]). To be specific, while the code is executed as shown in ① of Figure 7, the physical plan is constructed as usual by inserting the nodes into it. Then, upon the invocation of an action API (i.e., show function in line 5), the physical plan is optimized, and the final plan is relayed to the LAPUTAEXECUTOR to be processed further. We want to note that LAPUTA's libraries do not need to be protected as all the security critical operations are performed in the LAPUTAEXECUTOR.

LAPUTAEXECUTOR: Checking Plan and Executing Tasks. Upon receiving the final physical plan from LAPUTAPROCESSOR, LAPUTAEXECUTOR checks the owner provided policies against it (i.e., ② in Figure 7), and performs the task generation followed by task executions (i.e., ③ and ④). Meanwhile, LAPUTAEXECUTOR and the worker nodes run in the enclave (e.g., confidential VM of AMD SEV-SNP [5]) with encrypted channels so that the entire data analysis pipeline is protected. In addition, the data owners can attest that the correct LaputaChecker and Spark engine are loaded into LaputaExecutor, and then, they can securely send the data with the policies [5].

Firstly, LaputaExecutor filters out the plans that do not satisfy the policies using LaputaChecker (i.e., explained in §IV-A). Then, it generates the tasks from the validated plans using the Spark engine as usual. In this step, we can guarantee that the tasks are correctly generated as the execution is protected by confidential computing. Finally, LaputaExecutor distributes the tasks to the nodes after attesting them, and the actual data processing operations are performed, which are also safely protected in the enclaves. Thus, only the final results, which satisfy the policies as enforced by Laputa, are returned to the data users.

V. Implementation

We implemented Laputa on Spark [65] (v3.3) using Scala programming language (v2.12). We packaged Laputa's components (i.e., LaputaProcessor and LaputaExecutor) as extensible modules of Spark, and thus we did not intrusively modify the internal infrastructure of Spark. The entire implementation consists of ∼2600 lines of Scala code, and we explain the implementation details in following paragraphs.

We implemented Laputa's compartmentalized architecture using the (confidential) virtual machines on AMD SEV-ES enabled CPUs [5]. In particular, each VM installs the Spark engine, and one loads the LaputaProcessor module, and the other confidential one loads the LaputaExecutor module. As mentioned in §IV-B, the VM of LaputaProcessor runs the user applications and generates the physical plans at every invocation of `action` APIs. The confidential VM of LaputaExecutor is verified by the data owner and provisioned with sensitive data and the policies. On receiving physical plans, it executes them after checking if each plan adheres to owner policies, and sends back the results to the user application.

Our implemented modules contain the code to communicate between each one in each VM. For that, we used gRPC [26], where LaputaProcessor VM is the client and LaputaExecutor VM is the server. More specifically, communication is achieved as follows. First, the physical plans in the LaputaProcessor VM are intercepted by wrapping the Spark `action` API. Second, LaputaProcessor converts the `action` API into an RPC that terminates in the LaputaExecutor VM. To transmit the physical plan through RPC, we serialize its structure into a JSON format which includes all the information of the plan (e.g., operator class, expression class, tree hierarchy). Third, in the LaputaExecutor VM, the request is deserialized into a Spark physical plan and verified using the checker (i.e., LaputaChecker), which is implemented in-house using Scala. Finally, the validated physical plans are safely executed on the data and the analysis results are returned.

Our implementation has two limitations, none of which significantly impact our performance results. First, we do not currently implement remote attestation for the owner to attest the correctness of LaputaExecutor before sending data, but realizing the protocol is a matter of engineering. Second, a data owner's policy is pre-defined in a text file, which is loaded when the LaputaExecutor enclave is initialized. In the future, these policies can be dynamically loaded through network connections.

VI. Security Claims and Evaluation

In this section, we elaborate how Laputa guarantees that the policies are correctly enforced when retrieving the results from the owner's data (§VI-A). Then, we evaluate the correctness of LaputaChecker's implementation by enforcing synthetic policies (§VI-B) on TPC-H benchmark [51] with case studies (§VI-C).

A. *Claims on Security Guarantees of* Laputa

Laputa guarantees the owner provided policies are always enforced on data retrievals based on following two claims. Especially, the claims ensure that the data (and its descendent data flows) is always protected, and the malicious users (and cloud managers) cannot disclose the owner's data beyond what is defined in the owner's policy.

Claim 1. *Data users and cloud managers cannot directly access the owner's data.*

Owner's data is always encrypted at rest, in transit, and even in use as it can only be accessed through an enclave protected by confidential computing. In order to guarantee the data flows (not satisfying the policies) do not leave the protection domain, the owner initially attests the enclave loading LaputaExecutor before sending their data, and provides the encrypted data with encryption key only after the attestation succeeds. During the data analysis, all the distributed nodes run in the enclaves, and they perform mutual attestation against LaputaExecutor to get the encryption key for the owner's data [5], [35], which ensures the data is always encrypted throughout the data analysis procedure. After the execution, the distributed nodes terminate after safely erasing the encryption key.

Claim 2. *A malicious user cannot disclose the owner's data beyond what is defined in the owner's policy.*

After attesting the LaputaExecutor, the data owner registers the policies through an encrypted channel. Thus, it is impossible for the adversaries to tamper with the policies in transit. During the data analysis, Laputa mandates that only the authenticated driver node (i.e., node running LaputaExecutor) can schedule the tasks to be executed in the worker nodes—i.e., the authentication is performed through the attestation between the nodes. Thus, the physical plan constructed in untrusted Spark applications must always be relayed to the LaputaExecutor to be executed in the worker nodes. In the LaputaExecutor, security features of enclave guarantee that the plans are correctly checked by LaputaChecker and the tasks are correctly generated.

Table III: Description of the policies.

	Table	Policy Description
P1	all tables	Sensitive identifiers (i.e., primary keys and foreign keys) can only be used for joining tables.
P2	customer	Personally identifiable information (i.e., c_name, and c_acctval) must not be revealed.
P3	customer	Private information (i.e., c_mktsegment, c_comment, and c_phone) must not be obtained after filtering on PII (i.e., c_name, and c_acctval).
P4	customer	customer table must not be used in analytics alone.
P5	customer	Account balance (i.e., c_acctbal) should not be revealed directly or indirectly.
P6	customer	Phone number (i.e., c_phone) must always be used in conjunction with tail function.
P7	orders	If customer and orders tables are joined, address (i.e., c_address) must not be revealed with specific time (i.e., o_orderdate).

Table IV: Definition of the policies.

	Expression	Description
s_1	⟨Join,[keyList]⟩	Join using any of the sensitive identifiers fields (i.e., keyList)
s_2	⟨ALL$_{\mathtt{op}}$,[keyList,ALL$_{\mathtt{f}}$]⟩	Any with the sensitive identifiers
s_3	⟨Project,[{c_name,c_acctbal},ALL$_{\mathtt{f}}$]⟩	Project on personally identifiable information
s_4	⟨Project,[{c_mktsegment,c_comment,c_phone},ALL$_{\mathtt{f}}$]⟩	Project on private information
s_5	⟨Filter,[{c_name,c_acctbal},ALL$_{\mathtt{f}}$]⟩	Filter on personally identifiable information
s_6	⟨Join,[{c_custkey,c_nationkey},ALL$_{\mathtt{f}}$]⟩	Join on the keys of customer table
s_7	⟨Filter,[{c_acctbal},ALL$_{\mathtt{f}}$]⟩	Filter on the account balance with any filter condition.
s_8	⟨ALL$_{\mathtt{op}}$,[{c_phone},ALL$_{\mathtt{f}}$]⟩	All operators using phone number
s_9	⟨ALL$_{\mathtt{op}}$,[{c_phone},ALL$_{\mathtt{f}}$]⟩:(tail)	Any operators using phone number with tail function
s_{10}	⟨Join,[{o_custkey,c_custkey}]⟩	Join using the keys of orders and customer tables
s_{11}	⟨Filter,[{o_orderdate},ALL$_{\mathtt{f}}$]⟩	Filter on order date with any filter condition
s_{12}	⟨Project,[{c_address},ALL$_{\mathtt{f}}$]⟩	Project on the customer's address
P1	Allow ⟨AllTables, $(s_1 \mid {}^\wedge s_2)^*$⟩	Sensitive fields in keyList can only be used for join.
P2	Disallow ⟨customer, $.^*s_3$⟩	Personally identifiable information should not be directly projected at the final result.
P3	Disallow ⟨customer, $.^*s_5.^*s_4$⟩	When filtered on PII, private information must not be projected.
P4	Disallow ⟨customer, $({}^\wedge s_6)^*$⟩	customer table must not be used without join with other table.
P5	Disallow ⟨customer, $.^*s_7.^*$⟩	Filtering on account balance is always prohibited.
P6	Allow ⟨customer, $({}^\wedge s_8 \mid s_9)^*$⟩	Using phone number is allowed only when it is used with tail function.
P7	Disallow ⟨orders, $.^*s_{10}.^*s_{11}.^*s_{12} \mid .^*s_{11}.^*s_{10}.^*s_{12}$⟩	If customer and orders tables are joined, address is projected, filtering on the order date is prohibited.

Table V: List of filtered TPC-H queries by LAPUTACHECKER.

Query	Q2	Q3	Q5	Q8	Q10	Q11	Q15	Q18	Q22
Violated	P1	P1, P7	P7	P7	P1, P2, P6, P10	P1	P1	P1, P2, P10	P5

B. *Security Evaluation of* LAPUTACHECKER

We evaluate the correctness of LAPUTACHECKER's implementation by enforcing synthetic policies on TPC-H benchmark [51].

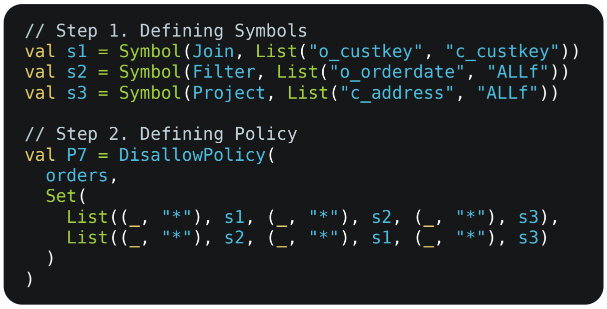
Benchmark and Policies. We leveraged TPC-H benchmark [51] as it is a well-known database benchmark with real world schema and complex queries. TPC-H benchmark consists of 8 tables and 22 SQL queries. Based on it, we came up with 7 synthetic policies (i.e., policy descriptions **P1** – **P7** as shown in Table III) with the goal of preventing the leakage of private customer information. In particular, amongst the 8 tables, customer table contains personal information of the customers (e.g., name, phone number, and account balance) and orders table contains the information of the orders submitted by the customers. Hence, we focused on preventing the extraction of private customer information from either of these tables. However, we want to note that defining the correct policies is beyond the scope of LAPUTA, and it is the responsibility of data owners.

Based on the policy descriptions, we defined the policies for LAPUTACHECKER as shown in **P1** – **P7** of Table IV. In order to define the policies, the owners first need to define the symbols as shown in the upper part of Table IV. Then, they can define the policies by combining the symbols like regular expression as shown in the lower part. Taking the policy **P6** as an example, the owners first need to define a symbol for checking whether the phone number (i.e., c_phone) is used, and if used, it is used with substring function. For that, we define the symbols $({}^\wedge s_8|s_9)$, which denotes the phone number is never used (i.e., ${}^\wedge s_8$), or it is used with substring function (i.e., s_9). Thus, combining this symbol into the pattern $({}^\wedge s_8|s_9)*$ indicates that the phone number is never used in the plan or it is used with substring function, satisfying the policy **P6**.

Results. Then, we ran the 22 queries in TPC-H benchmark on LAPUTA while enforcing the aforementioned policies. Amongst the 22 queries, we found that 9 queries violate one or more of our policies and are rejected by LAPUTACHECKER (i.e., Table V). We manually inspected the queries to confirm the correctness of this result, and no false-negatives were found—i.e., every query violating the policies were caught by LAPUTA. Please refer to §VIII for a full discussion on the soundness and completeness of LAPUTA's policy checking.

We briefly explain the violating queries below.

- Q2, Q11, and Q15 output the sensitive identifiers of the tables directly, violating **P1**.
- Q3 referenced sensitive identifiers and also used the filter operation with columns of customer table and orders table. Hence, it violate **P1**, and **P7**.
- Q5 and Q8 violate **P7** by filtering on o_orderdate and projecting c_address, when the customer and orders tables



```
// Step 1. Defining Symbols
val s1 = Symbol(Join, List("o_custkey", "c_custkey"))
val s2 = Symbol(Filter, List("o_orderdate", "ALLf"))
val s3 = Symbol(Project, List("c_address", "ALLf"))

// Step 2. Defining Policy
val P7 = DisallowPolicy(
  orders,
  Set(
    List((_, "*"), s1, (_, "*"), s2, (_, "*"), s3),
    List((_, "*"), s2, (_, "*"), s1, (_, "*"), s3)
  )
)
```

Figure 8: Scala code snippet for defining policy **P7**

are joined.

- Q10 and Q18 violate **P1**, **P2** and **P7**. It directly outputs the sensitive identifiers. Also, it outputs the columns of `customer` table which includes the name, address, and phone number of customer.
- Q10 violates **P6** as it uses phone number without `substring` function.
- Q22 was rejected due to filtering on personally identifiable information and projecting private information, violating **P5**. This query uses the `substring` function for `c_phone` but filters on `c_acctbal` field before projecting it.

C. *Case study*

In this section, we provide a real world example of defining a policy on LAPUTA, and how it is applied to the actual queries.

Defining Policy. We take the policy **P7** as an example. Policy **P7** prevents the users from leaking the customer's address histories while allowing them to analyze the customer and order information together. To this end, the owners can define the policy as shown in Figure 8. In particular, they should declare the symbols first using `Symbol` class that LAPUTA provides, and declare the policy (i.e., `AllowPolicy` or `DisallowPolicy` class) which takes the target table and the patterns.

Then, LAPUTACHECKER will inspect whether a physical plan constructed from the application matches one of the defined patterns $(*s_1 * s_2 * s_3)$ or $(*s_2 * s_1 * s_3)$. If the plan matches any of these patterns, it will abort the analysis, and inform that the plan violates the policy.

Query example. Once the owner completed defining the policies, the users will freely analyze the data as long as they satisfy the policies. We take the 10th query in TPC-H benchmark as an example:

```
Q10.
  select
    c_custkey, c_name,
    sum(l_extendedprice*(1-l_discount)) as revenue,
    c_acctbal,n_name,c_address,c_phone,c_comment
  from
    customer, orders, lineitem, nation
  where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= date '1993-10-01'
    and o_orderdate < date '1993-10-01' + interval '
    3' month
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
  group by
    c_custkey, c_name, c_acctbal, c_phone, n_name,
    c_address, c_comment
  order by
    revenue desc
  limit 20
```

This query violates **P1**, **P2**, **P6**, and **P7**. Taking the policy **P7** as an example, it first joins the `customer` and `orders` tables on the fields `c_custkey` and `o_custkey` (i.e., matching `s1` in previous code). Then, it filters the rows on `o_orderdate` (i.e., matching `s2`). Finally, it projects `c_address` as one of the output fields (i.e., matching `s3`), completing the pattern matching against **P7** in LAPUTACHECKER. Thus, the query is rejected.

VII. PERFORMANCE EVALUATION

This section evaluates the performance overhead of LAPUTA on database benchmarks and machine learning applications.

A. *Evaluation Settings*

Experimental Setup. We ran all the experiments on a cluster of three AMD machines, which have EPYC 7313 32 cores CPUs with 128 GiB RAM. All the machines run Ubuntu 22.04 image with Linux kernel v5.15.0. On top of that, we ran one virtual machine for each with 6 vCPUs and 16 GiB virtual memory to provide an equivalent environment for both the scenarios without and with confidential computing (i.e., AMD SEV-SNP [4]). In each VM, we ran LAPUTA which extends Spark v3.3.0.

Terminology. In order to clearly demonstrate the practical impact of LAPUTA, we evaluated the performance with three different settings **`Spark`**, **`Laputa-Insecure`**, and **`Laputa`**.

- **`Spark`** runs the native Spark [65] on a normal VM, not protected by AMD SEV-SNP. We use this setting as a baseline which shows the original performance without any policy enforcement and confidential computing.
- **`Laputa-Insecure`** enforces the policies as explained in §IV, but it does not use confidential computing. We use this setting to measure the overheads of LAPUTA's compartmentalization and policy check mechanism.
- **`Laputa`** is the same as **`Laputa-Insecure`** except it uses confidential computing (i.e., AMD SEV-SNP VMs). It is our proposed design that guarantees all the security properties.

Evaluation Targets. We evaluated each setting with following Spark applications (i.e., running SQL queries or machine learning models) with each database benchmark.

- **22 SQL queries on `TPC-H` benchmark** [51] are used to exhaustively measure the performance overhead of LAPUTA on various SQL queries. This benchmark is well known for evaluating the performance of database management systems [65], [22], [51].
- **3 SQL queries on `BDB` benchmark** [6] are used to measure the performance of LAPUTA on large-scale data processing and analytics. Each query is for measuring the time for filtering, aggregating, and joining large volume

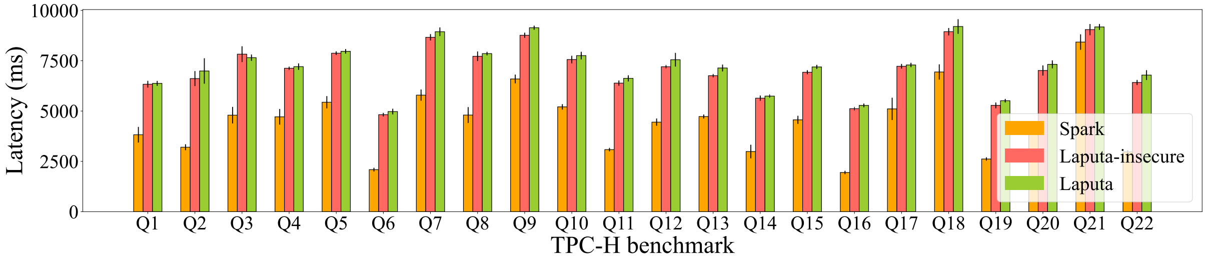


Figure 9: Increased latency on TPC-H benchmark.

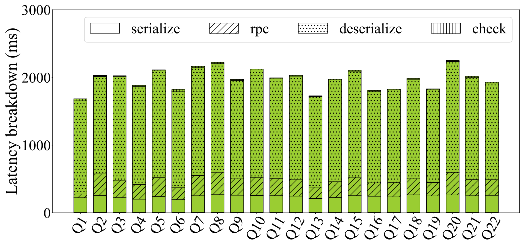


Figure 10: Latency breakdown of TPC-H benchmark.

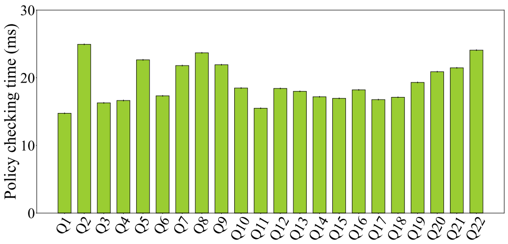


Figure 11: Policy checking time of TPC-H benchmark.

of data. We excluded fourth query in BDB benchmark as it uses external python scripts in query execution, which is currently not supported by LAPUTA.

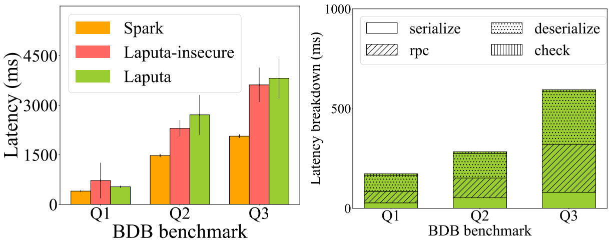
- **Recommendataion system on `Amazon product reviews`** [27], [34] is included to measure the overhead of LAPUTA on a real-world Spark application. The recommendation system is used to suggest a potentially purchasable products from the log of products that customers have reviewed. It uses recommendation model to analyze the reviews of the customers.
- **Clustering model on `Synthea health records`** [58] is also a real-world Spark application that applies K-means clustering on the dataset of patients' health records. It classifies the patients into various groups depending on their health records, which can be utilized by pharmaceutical companies and disease research groups [31].

B. Latency Overhead of LAPUTA

We measured the latency for handling each query and application in the benchmarks with three different settings—i.e., **`Spark`**, **`Laputa-Insecure`**, and **`Laputa`**.

22 SQL queries on `TPC-H` benchmark. As shown in Figure 9, **`Laputa`** has increased the latency about 35% for handling the queries in `TPC-H` benchmark. In particular, the latency increasement was mostly due to the LAPUTA's compartmentalization and policy check mechanism (i.e., increasing about 2500ms by **`Laputa-Insecure`**), and employing confidential computing has only increased 200 ms of latency on average (i.e., **`Laputa`** versus **`Laputa-Insecure`**).

We provide the breakdown of the latency increased by LAPUTA as shown in Figure 10. LAPUTA has shown fairly constant latency increasement across the queries, ranging from 1792 ms to 2460 ms. To be specific, the increased latency can be broken down into the time for serialization (to send the Spark physical plan over RPC), RPC transfer, deserialization, and policy checking. Especially, the deserialization takes much longer time than the serialization as LAPUTAEXECUTOR, which receives serialized physical plans, needs to load the module for deserializing the plan. Among them, the time for RPC transfer and deserialization were the dominant factors, which take about 81% of the increased latency. Since the deserialization takes more than 50% of the increased latency, we assume that optimizing this procedure would largely decrease the latency overhead of LAPUTA.

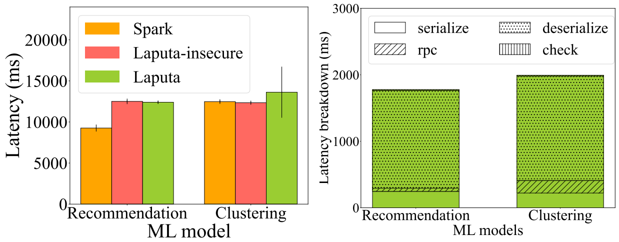


(a) Increased latency. **(b)** Latency breakdown.

Figure 12: Latency evaluation on `BDB` benchmark.

While the policy checking time takes only small portion, we show the time for clear demonstration (i.e., Figure 11). LAPUTA consumes longer time to check the policies as more complex queries are handled—i.e., queries `Q8` and `Q22` are the most complex queries in `TPC-H` benchmark, which joins 8 tables and contains multiple subqueries. However, we want to note that the policy checking does not largely affect the latency, as it occupies only 0.68% of the increased latency. While, it is possible for the policy checking to take much longer time due to a catastrophic backtracking [13], there is a little chance for a Spark physical plan to have such complex pattern.

3 SQL queries on `BDB` benchmark. LAPUTA shows a similar tendency on `BDB` benchmark as it does on the `TPC-H` benchmark, as illustrated in Figure 12-(a). As expected, Figure 12-(b) demonstrates that the time for serialization, RPC transfer, and



(a) Increased latency.

(b) Latency breakdown.

Figure 13: Latency evaluation on ML models.

deserialization increases as the query becomes larger—i.e., from the query `Q1` to `Q3`. To be specific, each query `Q1`, `Q2`, and `Q3` has 30, 40, and 60 LoC of Scala, respectively.

Real world applications using ML models. We also conducted the evaluation on real world Spark applications implementing recommendation system, and K-means clustering. As shown in Figure 13, LAPUTA shows consistent tendency as explained in previous experiments.

C. Throughput Overhead of LAPUTA

Then, we measured the throughput of each setting while varying the number of concurrent requests.

22 SQL queries on `TPC-H` benchmark. In `TPC-H` benchmark, we measured the throughput with 64 concurrent queries, and LAPUTA shows almost comparable throughput as the baseline Spark as shown in Figure 14. Overhead introduced by LAPUTA is minimized as the time for those operations is overlapped with the time for executing the other queries.

3 SQL queries on `BDB` benchmark. As shown in Figure 15-(a), LAPUTA has decreased 38% throughput on average compared to the baseline Spark setting. Especially, LAPUTA affects the most on the query `Q1`, as it has larger overhead to serialize, transfer, and deserialize the physical plan, while the time for executing the plan is relatively small.

Real world applications using ML models. On the other hand, LAPUTA shows comparable throughput performance when it comes to the Spark applications using ML models (i.e., Figure 15-(b)). This is because those applications take longer time to execute the plans compared to the time for LAPUTA's logic.

D. Takeaway of LAPUTA *Adoption Cost*

Overall, LAPUTA imposes about 35% of latency and 25% of throughput overheads on average, where the compartmentalization overhead occupies the majority. While there are faster alternative compartmentalization mechanisms (e.g., in-process isolation [53]), these are not compatible with our threat model of cloud computing. Thus, LAPUTA employs VM-based confidential computing, which is becoming the standard approach to construct trusted computing environments in clouds [11]. However, this overhead would still be acceptable in areas that prioritize stronger security enforcement (e.g., medical and financial data sharing [31], [1]).

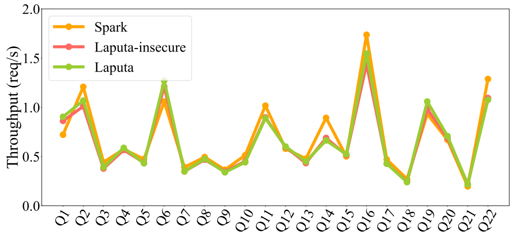
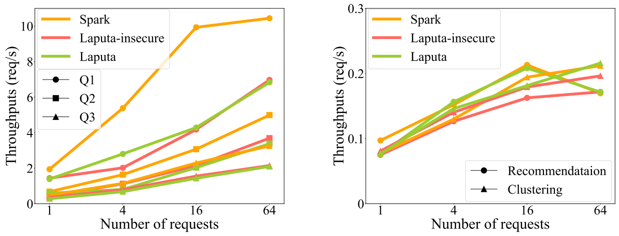


Figure 14: Throughput evaluation on `TPC-H` benchmark with 64 concurrent queries.



(a) `BDB` benchmark.

(b) ML models.

Figure 15: Throughput evaluation on `BDB` benchmark and ML models.

VIII. DISCUSSION

Expressiveness of LAPUTA's Policy Language. The expressiveness of LAPUTA's policy language depends on i) the expressiveness of regular expression (i.e., pattern), and ii) the definition of symbols.

First, data owners cannot express policies that require constructs outside regular languages [64]. While most policies can be constructed within regular languages, there can be some unusual policies. For example, LAPUTA does not allow the expression of a plan that contains a number of `Join` followed by the same number of `Filter`, as the pattern for such plans (i.e., $\{s^n_{\texttt{Join}} s^n_{\texttt{Filter}} | n >= 1\}$) is not a regular language.

Second, LAPUTA cannot express a node that uses multiple fields together without introducing false-positives. For example, in order to express a `Filter` node using `age` and `gender` together in its condition, data owners must define a symbol $s = \langle \texttt{Filter}, [\{\texttt{age}, \texttt{gender}\}] \rangle$. However, such symbol has false-positives as any `Filter` node using only `age` or `gender` can also be matched as well as the correct nodes using both fields. In addition, LAPUTA cannot match a node using user-defined functions (UDFs) in its expression, as LAPUTA currently does not support using UDF in defining the symbols. However, we can resolve these limitations by improving the symbol definition—i.e., explained more in §A.

Despite aforementioned limitations, LAPUTA significantly improves over prior work by providing a way to express (and categorize) fine-grained Spark physical plans, while prior work is limited to pre-determined rules [37], [60], [61]. Using Qapla's linking policy [37] as an example, which prevents using both fields (e.g., `X`, and `Y`) in the same plan, LAPUTA can enforce such policy by allowing only the plans matching a pattern $(^\wedge s_{\texttt{X}})^*|(^\wedge s_{\texttt{Y}})^*$, where $s_{\texttt{X}}$ and $s_{\texttt{Y}}$ match any node using either field `X` or `Y`. Policies of SparkAC [60] and GuardSpark++[61],

Table VI: Comparison of the expressiveness of Laputa's policy against previous works. **`Single col.`**: can express a node using single column—e.g., `Filter(disease)`? **`Multi cols.`**: can express a node using multiple columns—e.g., `Project(name, disease)`? **`Plan structure`**: can express the structure of the paln—e.g., `Join` after `Filter`? **`Time related`**: can express queries over time—e.g., 10 queries in 1 second? **`Any ops&cols.`**: can express a node using any operator with any columns—e.g., `Kmeans(name,any)`?

Capability	`Single col.`	`Multi cols.`	`Plan structure`	`Time related`	`Any ops&cols.`
sparkAC [60]	Yes	No	Limited	No	No
GuardSpark [61]	Yes	No	Limited	No	No
Qapla [37]	Yes	Yes	No	No	No
Datalawyer [52]	Yes	Yes	No	Yes	No
Laputa	**Yes**	**Yes**	**Yes**	**No**	**Yes**

which block using certain fields with a specific operator, can be easily expressed in Laputa by defining the symbols. However, those works cannot express the policy $\mathbf{P_3}$ in §III (i.e., filter cannot be applied on a specific field if the table is joined), while Laputa can.

Soundness and Completeness of Laputa's Policy Checking. For policies that can be expressed by data owners (explained in the previous heading), Laputa's policy checking mechanism is both sound and complete. The intuition behind this lies in the fact that a data owner's policy can be precisely pattern-matched into physical plans created by Spark. Therefore, by design, all plans matching the patterns are caught by Laputa (i.e., no false-negatives), and all plans not matching the patterns are not caught by Laputa (i.e., no false-positives).

Complexity of Defining Policy. Laputa provides more flexible way to define the policies at the expense of increased complexity. In order to define the policies in Laputa, the data owner has to understand the operators in Spark, the structure of the physical plans, and the regular expression. However, much of the complexity can be avoided by pre-determining commonly used symbols and patterns (e.g., $.*s_{\mathtt{X}}$ for matching the plans projecting on `X`).

Potential Covert Channel and Mitigation. While Laputa prevents direct data breaches, a malicious data user may construct a covert channel on Laputa to leak the auxiliary information of database. For example, joining after filtering on a specific disease may leak the number of diagnosed patients through timing side channel as the query latency depends on the number of rows. In order to prevent such attacks, Laputa can employ well-known side channel mitigation techniques such as constant time execution [24], or oblivious computation [2].

IX. Related works

Policy Enforcement in Data Analytics. Several previous works have studied on enforcing policies while analyzing database [52], [37], [60], [61]. While each work designs different mechanisms to enforce the policies, we summarize the high-level comparison of those works against Laputa in Figure VI. In particular, we categorize the expressability of the policies into the usage of single column (i.e., `Single col.`), the usage of multiple columns simultaneously (i.e., `Multi cols.`), structure of the plan (i.e., `Plan structure`), queries over time (i.e., `Time related`), and the usage of any operators with any columns (i.e., `Any ops&cols.`).

SparkAC [60] and GuardSpark++ [61] design an access control mechanism on Spark, which extends purpose based access control [14]. Specifically, these works categorize each operator based on the purpose (e.g., `Filter` and `Scan` for retrieving data), and determine whether each column can be used for each purpose (i.e., supporting `Single col.`). However, they do not consider using multiple columns simultaneously (i.e., not supporting `Multi cols.`), and matching plan structure.

Qapla [37] provides a predefined set of rules to filter out the SQL queries. It supports expressing the usage of single column, and multiple columns, but the available operators are limited—e.g., it cannot prevent using multiple columns for only a specific operator. Furthermore, it does not capture the structure of the plan, limited in expressing complex requirements. We want to note that Laputa can express all the rules defined in Qapla.

DataLaywer [52] designs a mechanism to express the policies using SQL, which specifically supports defining the policies related to time. However, it has limited support for expressing the structure of the query plans and various operators.

Compartmentalization of Spark Engine. Similar to Laputa, SparkConnect [19] compartmentalizes Spark applications as well. The key difference is that SparkConnect is designed to provide better modularity for development. As a result, it does not securely enforce data usage policies on the physical plans. Thus, SparkConnect still needs a mechanism like LaputaChecker to enforce the policies. Regarding the compartment design, Laputa offers two additional security benefits over SparkConnect, because it compartmentalizes the application at a physical plan layer, whereas SparkConnect does so at an earlier logical plan layer. First, Laputa minimizes the TCB, as only the core Spark logic for executing a physical plan runs in the enclave compartment. Second, Laputa offers security-oriented maintenance, as most of the Spark logic (e.g., plan optimizer) can be updated without modifying the enclave's component, avoiding the risk of introducing bugs into security critical logics.

Data Analytics on Confidential Computing. Opaque [66] and OCQ [20] use confidential computing to protect the data from malicious cloud providers. However, they do not design policy enforcement mechanism, as they trust data users. On the other hand, Laputa rearchitects the Spark applications, and employs policy enforcement to protect the data from untrusted data users. Ryoan [28] designs a system to protect distributed applications on hardware enclaves, but it does not focus on protecting data from malicious data users.

X. Conclusion

In this paper, we propose LAPUTA, a mechanism for secure policy enforcement on Spark platform. LAPUTA provides a new pattern matching based policy enforcement which uses fine-grained policies on Spark physical plans. In addition, LAPUTA designs confidential computing based compartmentalization to protect the data analysis pipeline from malicious Spark applications and cloud infrastructure. We implemented the prototype of LAPUTA and demonstrated that LAPUTA correctly enforces policies with moderate performance overhead.

XI. Acknowledgment

We would like to thank anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093). This research was supported by Start up Pioneering in Research and Innovation(SPRINT) through the Commercialization Promotion Agency for R&D Outcomes(COMPA) grant funded by the Korea government(Ministry of Science and ICT) (No. 2710002717). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work. Adil Ahmad was partly supported by the US Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0204 during the course of this work. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not reflect the views of AFOSR.

References

[1] Emmanuel A Abbe, Amir E Khandani, and Andrew W Lo. Privacy-preserving methods for sharing financial risk exposures. *American Economic Review*, 102(3):65–70, 2012.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[3] Muhammad Ali and Khurshed Iqbal. The role of apache hadoop and spark in revolutionizing financial data management and analysis: A comparative study. *Journal of Artificial Intelligence and Machine Learning in Management*, 6(2):14–28, 2022.

[4] Advanced Micro Devices (AMD). AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020.

[5] Advanced Micro Devices (AMD). AMD Memory Encryption. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf, 2021.

[6] amplab. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2014.

[7] George J Annas. Hipaa regulations: a new era of medical-record privacy? *New England Journal of Medicine*, 348:1486, 2003.

[8] Niyazi Ari and Makhamadsulton Ustazhanov. Matplotlib in python. In *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*, pages 1–6. IEEE, 2014.

[9] Arm. Arm Security Technology - Building a Secure System using TrustZone Technology, 2005.

[10] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[11] Microsoft Azure. About Azure Confidential VMs. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview, 2024.

[12] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. Smcql: secure querying for federated databases. In *VLDB Endowment*, volume 10, 2017.

[13] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. *arXiv preprint arXiv:1405.5599*, 2014.

[14] Ji-Won Byun, Elisa Bertino, and Ninghui Li. Purpose based access control of complex data for privacy protection. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 102–110, 2005.

[15] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 253–264, 2013.

[16] CNN. Meta agrees to pay $725 million to settle lawsuit over Cambridge Analytica data leak. https://edition.cnn.com/2022/12/23/tech/meta-cambridge-analytica-settlement/index.html, 2022.

[17] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007.

[18] databricks. Introducing Apache Spark 3.0. https://www.databricks.com/blog/2020/06/18/introducing-apache-spark-3-0-now-available-in-databricks-runtime-7-0.html, 2020.

[19] DataBricks. Spark connect overview. https://spark.apache.org/docs/latest/spark-connect-overview.html, 2024.

[20] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[21] Edward S Dove and Mark Phillips. Privacy law, data sharing policies, and medical data: a comparative perspective. *Medical data privacy handbook*, pages 639–678, 2015.

[22] Leila Etaati and Leila Etaati. Azure databricks. *Machine Learning with Microsoft Technologies: Selecting the Right Architecture and Tools for Your Project*, pages 159–171, 2019.

[23] Janice Feinberg and L Michael Posey. Companies market hospital drug-use data 1622 to the pharmaceutical industry. *American Journal of Hospital Pharmacy*, 43, 1986.

[24] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *information retrieval*, 4:133–151, 2001.

[25] Eric Goldman. An introduction to the california consumer privacy act (ccpa). *Santa Clara Univ. Legal Studies Research Paper*, 2020.

[26] Google. gRPC. https://grpc.io/.

[27] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class co@miscnvidia2022h100, title=H100 tensor core GPU architecture overview, author=NVIDIA, NVIDIA, year=2022 llaborative filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517, 2016.

[28] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[29] Intel. Intel Trusted Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[30] Changqing Ji, Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. Big data processing in cloud computing environments. In *2012 12th international symposium on pervasive systems, algorithms and networks*, pages 17–23. IEEE, 2012.

[31] Hao Jin, Yan Luo, Peilong Li, and Jomol Mathew. A review of secure and privacy-preserving medical data sharing. *IEEE Access*, 7:61656–61669, 2019.

[32] Sunitha Kambhampati. How to extend Spark with customized optimizations. https://hyperj.net/note.arts/asset/pdf/how-to-extend-apache-spark-with-customized-optimizations.pdf, 2019.

[33] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Bertty Contreras-Rojas, Rodrigo Pardo-Meza, Anis Troudi, and Sanjay Chawla. Ml-based cross-platform query optimization. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1489–1500. IEEE, 2020.

[34] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 43–52, 2015.

[35] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isoalted execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, 2013.

[36] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[37] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1463–1479, 2017.

[38] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The journal of machine learning research*, 17(1):1235–1241, 2016.

[39] Microsoft. Azure Synapse Analytics home page. https://azure.microsoft.com/en-us/products/synapse-analytics/.

[40] Robert L. Mitchell. Medical data sharing: Are we there yet? https://hyperj.net/note.arts/asset/pdf/how-to-extend-apache-spark-with-customized-optimizations.pdfhttps://www.computerworld.com/article/3702109/medical-data-sharing-electronic-health-record-exchanges.html, 2023.

[41] Shi Na, Liu Xumin, and Guan Yong. Research on k-means clustering algorithm: An improved k-means clustering algorithm. In *2010 Third International Symposium on intelligent information technology and security informatics*, pages 63–67. Ieee, 2010.

[42] NVIDIA NVIDIA. H100 tensor core gpu architecture overview, 2022.

[43] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[44] Rahul Potharaju, Terry Kim, Eunjin Song, Wentao Wu, Lev Novik, Apoorve Dave, Andrew Fogarty, Pouria Pirzadeh, Vidip Acharya, Gurleen Dhody, et al. Hyperspace: the indexing subsystem of azure synapse. *Proceedings of the VLDB Endowment*, 14(12):3043–3055, 2021.

[45] General Data Protection Regulation. General data protection regulation (gdpr). *Intersoft Consulting, Accessed in October*, 24(1), 2018.

[46] Abhishek Roy, Alekh Jindal, Priyanka Gomatam, Xiating Ouyang, Ashit Gosalia, Nishkam Ravi, Swinky Mann, and Prakhar Jain. Sparkcruise: workload optimization in managed spark clusters at microsoft. *Proceedings of the VLDB Endowment*, 14(12):3122–3134, 2021.

[47] Christoph L Schuba, Ivan V Krsul, Markus G Kuhn, Eugene H Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on tcp. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*, pages 208–223. IEEE, 1997.

[48] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[49] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[50] The Fintech Times. Open Banking: Data Sharing Vs Payment Services Approach. https://thefintechtimes.com/open-banking-data-sharing-vs-payment-services-approach/, 2023.

[51] TPC. TPC-H Benchmark. https://www.tpc.org/tpch/, 2023.

[52] Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu. Automatic enforcement of data use policies with datalawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 213–225, 2015.

[53] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({{{{{MPK}}}}}). In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings fo the 27th USENIX Security Symposium*. USENIX Association, 2018.

[55] Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36. Santa Clara, CA, 2007.

[56] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[57] Bill Venners. The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.

[58] Jason Walonoski, Mark Kramer, Joseph Nichols, Andre Quina, Chris Moesel, Dylan Hall, Carlton Duffett, Kudakwashe Dube, Thomas Gallagher, and Scott McLachlan. Synthea: An approach, method, and software mechanism for generating synthetic patients and the synthetic electronic health care record. *Journal of the American Medical Informatics Association*, 25(3):230–238, 2018.

[59] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 640–656. IEEE, 2015.

[60] Tao Xue, Yu Wen, Bo Luo, Gang Li, Yingjiu Li, Boyang Zhang, Yang Zheng, Yanfei Hu, and Dan Meng. Sparkac: Fine-grained access control in spark for secure data sharing and analytics. *IEEE Transactions on Dependable and Secure Computing*, 20(2):1104–1123, 2022.

[61] Tao Xue, Yu Wen, Bo Luo, Boyang Zhang, Yang Zheng, Yanfei Hu, Yingjiu Li, Gang Li, and Dan Meng. Guardspark++: Fine-grained purpose-aware access control for secure data sharing and analysis in spark. In *Annual Computer Security Applications Conference*, pages 582–596, 2020.

[62] Chaowei Yang, Qunying Huang, Zhenlong Li, Kai Liu, and Fei Hu. Big data and cloud computing: innovation opportunities and challenges. *International Journal of Digital Earth*, 10(1):13–53, 2017.

[63] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[64] Fang Yu, Zhifeng Chen, Yanlei Diao, Tamil V Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, 2006.

[65] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[66] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, volume 17, pages 283–298, 2017.

APPENDIX

This section provides a formal description of LAPUTA's policy language and explains how a Spark physical plan is matched against the defined policies.

Formal Description of LAPUTA's Policy Language. We provide the preliminaries of LAPUTA's policy language as shown in Table VII. In order to define the policies in LAPUTA, data owners first define the symbols as follows:

- Symbol $s = \langle o, A \rangle : (e)$, where o denotes a Spark logical operator. For this, LAPUTA accepts the operators listed in Table VIII or a wildcard operator ($\texttt{ALL}_{\texttt{op}}$) introduced to match any operators. The second argument (A) denotes a set that takes a subset of all fields in table t (i.e., F_t) as an element—i.e., $A \subseteq \mathcal{P}(F_t)$[2], and an optional argument (e) denotes an expression constructed from the fields used in A.

Given the user-defined symbols $s_1, ..., s_N$, the set of symbols Σ is defined as follows:

- Symbols $\Sigma = \{s_1, ..., s_N\} \cup \{\alpha\}$, where a symbol s_i is defined by the data owner, and a hidden symbol α is introduced to denote the nodes (in the Spark physical plan) that are not denoted by any of the user-defined symbols $s_1, ..., s_N$.

Then, LAPUTA's policy P is defined as a regular language over the set of symbols Σ.

- Policy $P = \langle t, r \rangle$, where t denotes a table (i.e., leaf node in the Spark physical plan), and r is a regular expression built from a set of symbols Σ.

LAPUTA's Interface for Defining Symbols. However, it is cumbersome to define the set A by manually adding all available subsets. Thus, LAPUTA provides a function $[\cdot]$ to help define the set A, which takes a set of fields as input and returns all subsets of the given set except the empty set (i.e., available fields as described in §IV-A):

- $[X] := \{X' | X' \subseteq X \ \wedge \ X' \neq \phi\}$, which represents any subset of $X (\subseteq F_t)$ except an empty set.

For example, $[\{\texttt{gender}, \texttt{age}\}]$ contains any sets that can be constructed using either `gender` or `age` (i.e., $\{\{\texttt{gender}\}, \{\texttt{age}\}, \{\texttt{gender}, \texttt{age}\}\}$). Using the function $[\cdot]$, usage of any fields can be expressed as $[F_t]$, which is denoted as a wildcard symbol $\texttt{ALL}_{\texttt{f}}$. However, using the function $[\cdot]$ alone cannot express the inclusive sets that contain a given field as well as any other fields (e.g., any subset of $\{\texttt{name}, \texttt{gender}, \texttt{age}\}$ that contains `age`). To this end, LAPUTA provides another function $[\cdot, \texttt{ALL}_{\texttt{f}}]$, which returns any subsets containing the given fields:

- $[X, \texttt{ALL}_{\texttt{f}}] := \texttt{ALL}_{\texttt{f}} - [X^c]$, which represents any subsets that contain at least one field in X.

For example, $[\{\texttt{age}\}, \texttt{ALL}_{\texttt{f}}]$ is $\texttt{ALL}_{\texttt{f}} - [\{\texttt{age}\}^c] = \texttt{ALL}_{\texttt{f}} - [\{\texttt{name}, \texttt{gender}\}]$, which excludes subsets not containing `age` from entire subsets, thereby representing any subsets containing `age`.

Table VII: Preliminaries of LAPUTA's policy language.

Note	Definition	Example
$\boldsymbol{o}$	Spark logical operator (i.e., Table VIII)	`Filter`
$\boldsymbol{A}$	Set of subsets of the fields in table	$\{\{\texttt{gender}\}, \{\texttt{age}\}, \{\texttt{gender}, \texttt{age}\}\}$
$\boldsymbol{e}$	Optional expression	`gender == MALE && age > 20`
$\boldsymbol{s}$	User-defined symbol (i.e., $s = \langle o, A \rangle : (e)$)	$s_1 = \langle \texttt{Filter}, \{\{\texttt{age}\}\} \rangle : (\texttt{age > 20})$ $s_2 = \langle \texttt{Project}, [\{\texttt{name}, \texttt{gender}\}] \rangle$
$\boldsymbol{\alpha}$	Hidden symbol	-
$\boldsymbol{t}$	Database table F_t denoting all fields of t	Table t_1 of `name`, `gender`, and `age` $F_{t_1} = \{\texttt{name}, \texttt{gender}, \texttt{age}\}$
$\boldsymbol{r}$	Regular expression of symbols	$\texttt{.*}s_1\texttt{.*}s_2$
$\boldsymbol{P}$	Policy (i.e., $P = \langle t, r \rangle$)	$P_1 = \langle t_1, \texttt{.*}s_1\texttt{.*}s_2 \rangle$

Using the functions $[\cdot]$ and $[\cdot, \texttt{ALL}_{\texttt{f}}]$, we can express comprehensive combinations of the available fields (e.g., using either `gender`, `age`, or both together). However, we cannot express conjunctive combinations that require both fields to be included together (e.g., `gender` and `age` only). Nevertheless, we can extend the functions based on set operations to express such cases (e.g., $\{\{\texttt{gender}, \texttt{age}\}\}$ is $[\{\texttt{gender}, \texttt{age}\}] - [\{\texttt{gender}\}] - [\{\texttt{age}\}]$).

For the final optional argument e, LAPUTA accepts an expression composed of Scala and Spark built-in functions, using the fields specified in A. Thus, the argument e follows the same syntax of Scala expressions that can be constructed using arithmetic operations as well as other built-in functions (e.g., `gender == MALE && age > 20`). While current LAPUTA's implementation does not allow user-defined functions (UDFs), it can be extended to incorporate the UDF implementation into Spark physical plans and check them against the given expression e.

LAPUTA's Policy Checking Mechanism. LAPUTA checks whether Spark physical plans (provided by data users) satisfy the policies (given by the data owner). To this end, LAPUTA i) splits the Spark physical plan (i.e., DAG of nodes) into the sequences of nodes for each table (explained in §IV-B), ii) converts each node in the sequence to the corresponding symbol (thereby getting a sequence of symbols), and finally, iii) matches the sequence of symbols against the given regular expression (in policy) following the common regular expression matching algorithms [49]. The first and last steps are straightforward, thus, we next explain how LAPUTA converts a node (in a plan) to the symbol (defined by data owner).

LAPUTA defines a many-to-one mapping function, which converts a node (composed of Spark physical operator and expressions using the table's fields) to one of the user-defined symbols (i.e., s_i or α in Σ):

- A node n is converted to a symbol s_i $(= \langle o, A \rangle : (e))$, if
  i. Physical operator in the node matches the logical operator o as illustrated in Table VIII.
  ii. Set of fields used in the node belongs to the set A (i.e., collection of subsets of the fields).
  iii. Expressions in the node match the optional expression e as illustrated in Table IX.

[2] $\mathcal{P}$: powerset—i.e., set of all possible subsets

Table VIII: A list of Spark logical operators that can be used to define a symbol.

Logical operators	Physical operators	Description
`Except()`	`ExceptExec`	Return rows in the first input table that are not in the second table.
`Intersect()`	`IntersectExec`	Return rows that are common in both input tables.
`Union()`	`UnionExec`	Combine rows from both input tables.
`Limit(`**n**`)`	`GlobalLimitExec` `LocalLimitExec`	Return first **n** rows from input table.
`Offset(`**n**`)`	`TakeOrderedAndProjectExec`	Skip first **n** rows and return the rest.
`Tail(`**n**`)`	`TailExec`	Return the last **n** rows.
`Sample(`**n**`)`	`SampleExec`	Randomly sample and return **n** rows.
`Sort(`**e**`)`	`SortExec`	Sort rows by expression **e**.
`Filter(`**e**`)`	`FilterExec`	Filter rows with condition **e**.
`Expand(`**e**`)`	`ExpandExec`	Generate multiple rows from the result of **e**.

`Join(`$\mathbf{e}_1$`,`$\mathbf{e}_2$`)`	`BroadcastHashJoinExec`
	`SortMergeJoinExec`
	`ShuffleHashJoinExec`
	`BroadcastNestedLoopJoinExec`
		Join two tables by $\mathbf{e}_1$==$\mathbf{e}_2$, where $\mathbf{e}_1$ is constructed from the fields of first table, and $\mathbf{e}_2$ is constructed from the fields of second table.

`Aggregate(`$\mathbf{e}_1$`,`$\mathbf{e}_2$`)`	`HashAggregateExec` `SortAggregateExec`	Aggregate rows after applying $\mathbf{e}_2$, grouped by $\mathbf{e}_1$.
`Window(`$\mathbf{e}_1$`,`$\mathbf{e}_2$`,`$\mathbf{e}_3$`)`	`WindowExec`	Grouping multiple rows using $\mathbf{e}_1$, and apply $\mathbf{e}_3$ after sorting by $\mathbf{e}_2$.
`Project(`$\mathbf{e}_1$`,`$\mathbf{e}_2$`,...)`	`ProjectExec`	Print rows after applying $\mathbf{e}_1$,$\mathbf{e}_2$,... to the table.
$\mathtt{ALL}_{\mathtt{op}}$	Any physical operators	Wildcard operator to match any Spark physical operators.

n denotes an integer. **e** denotes an expression using fields of the table.

Table IX: Expression matching rules depending on the operators.

Number of expressions	Operators	Default (no expression)	Specified

0	`Except`
	`Intersect`
	`Union`
	`Limit`
	`Offset`
	`Tail`
	`Sample`
		not applicable	

1	`Sort`
	`Filter`
	`Expand`
		any expression	only specified expression

	`Join`*	**only identity**	**not allow specifying**
2	`Aggregate`	$\mathbf{e}_1$: only identity $\mathbf{e}_2$: any expression	$\mathbf{e}_1$: only identity $\mathbf{e}_2$: only specified expression
3	`Window`	$\mathbf{e}_1$: only identity $\mathbf{e}_2$: only identity $\mathbf{e}_3$: any expression	$\mathbf{e}_1$: only identity $\mathbf{e}_2$: only identity $\mathbf{e}_3$: only specified expression
Many	`Project`	any expression	applied only to fields used in optional expression
	$\mathtt{ALL}_{\mathtt{op}}$	any expression	allow only specifying identity

$*$: `Join` accepts only one expression per joined table.

If a node does not satisfy the coversion criteria for any defined symbol, LAPUTA returns α.

In some cases, multiple symbols may satisfy the conditions for a given node—e.g., if $s_1 = \langle \mathtt{Filter}, \mathtt{ALL}_{\mathtt{f}} \rangle$, and $s_2 = \langle \mathtt{Filter}, [\{\mathtt{age}\}] \rangle$, a node `FilterExec(age > 20)` matches both symbols. Thus, LAPUTA provides priority rules as follows to return only one symbol at a time:

- When multiple symbols match the conditions, the most appropriate symbol is returned following

i. Symbols with an exact operator take priority over the symbols with $\mathtt{ALL}_{\mathtt{op}}$.

ii. Symbols with a smaller size of set A take priority over the others.

iii. Symbols with an optional expression e take priority over the others.

Thus, the node `FilterExec(age > 20)` in the above example will be converted to the symbol s_2 as the symbol s_2 has smaller set A (i.e., $\{\{\mathtt{age}\}\}$) than the symbol s_1 (i.e., $\mathtt{ALL}_{\mathtt{f}} = [F_t]$). LAPUTA aborts with an error if multiple symbols remain even after applying these priority rules.