KAKUTE: A Precise, Unified Information Flow Analysis System for Big-data Security

Jianyu Jiang
The University of Hong Kong
jyjiang@cs.hku.hk

Shixiong Zhao
The University of Hong Kong
sxzhao@connect.hku.hk

Danish Alsayed
The University of Hong Kong
dibrahim@connect.hku.hk

Yuexuan Wang
Zhejiang University
The University of Hong Kong
amywang@hku.hk

Heming Cui
The University of Hong Kong
heming@cs.hku.hk

Feng Liang
The University of Hong Kong
loengf@connect.hku.hk

Zhaoquan Gu
Guangzhou University
The University of Hong Kong
guzhaoquan@126.com

ABSTRACT

Big-data frameworks (e.g., Spark) enable computations on tremendous data records generated by third parties, causing various security and reliability problems such as information leakage and programming bugs. Existing systems for big-data security (e.g., Titian) track data transformations in a record level, so they are imprecise and too coarse-grained for these problems. For instance, when we ran Titian to drill down input records that produced a buggy output record, Titian reported 3 to 9 orders of magnitude more input records than the actual ones. Information Flow Tracking (IFT) is a conventional approach for precise information control. However, extant IFT systems are neither efficient nor complete for big-data frameworks, because theses frameworks are data-intensive, and data flowing across hosts is often ignored by IFT.

This paper presents KAKUTE, the first precise, fine-grained information flow analysis system for big-data. Our insight on making IFT efficient is that most fields in a data record often have the same IFT tags, and we present two new efficient techniques called Reference Propagation and Tag Sharing. In addition, we design an efficient, complete cross-host information flow propagation approach. Evaluation on seven diverse big-data programs (e.g., WordCount) shows that KAKUTE had merely 32.3% overhead on average even when fine-grained information control was enabled. Compared with Titian, KAKUTE precisely drilled down the actual bug inducing input records, a huge reduction of 3 to 9 orders of magnitude. KAKUTE's performance overhead is comparable with Titian. Furthermore, KAKUTE effectively detected 13 real-world security and reliability bugs in 4 diverse problems, including information leakage, data provenance, programming and performance bugs. KAKUTE's source code and results are available on https://github.com/hku-systems/kakute.

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134607

CCS CONCEPTS

• **Security and privacy**; • **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

Information Flow Tracking, Data-intensive Scalable Computing System, Big-data

ACM Reference Format:
Jianyu Jiang, Shixiong Zhao, Danish Alsayed, Yuexuan Wang, Heming Cui, Feng Liang, and Zhaoquan Gu. 2017. KAKUTE: A Precise, Unified Information Flow Analysis System for Big-data Security. In *Proceedings of ACSAC 2017*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3134600.3134607

1 INTRODUCTION

Data-Intensive Scalable Computation (DISC) frameworks (e.g., Spark [43]) enable computation on enormous data records in various domains, including e-business, finance, medical analysis and military. These frameworks allow people to write user-defined-functions (UDFs) to process data from third parties, which can cause serious problems such as sensitive information leakage and programming bugs. Tackling these problems needs fine-grained tracking of data. For example, in an eBay order record ⟨`time`, `creditCardId`, `productID`⟩, the `creditCardId` field is sensitive and it must not be leaked in UDFs.

To address these problems, extant systems (e.g., Titian [19]) adopt a record-level tracking approach. This approach stores mappings of input and output records in each transformation (e.g., `map` and `groupByKey`). Transformations of records (or *lineage*) are obtained with a simple recursive traversal of the mapping tables. By doing so, given a buggy output record, input records that produce the output can be identified, the so-called *data provenance* problem [19]. Data provenance with high precision (i.e., identifying only bug-inducing inputs) is crucial for many security problems, including post-incident investigation [2], integrity verification [27] and privacy control [26].

Unfortunately, this record-level tracking approach has two major issues. First, it is imprecise for provenance, as it will return many irrelevant records, especially for programs with many-to-many transformations (e.g., `groupByKey`). DISC systems treat many-to-many transformations as black boxes as these transformations usually contain UDFs. As a result, all inputs that go through a transformation will be returned when back-tracing one output (§3.3). Second, record-level tracking is too coarse-grained. For example, a small portion of fields in a record may have sensitive tags, and tracing at field-level granularity is necessary for information control [1].

Conventionally, Information Flow Tracking (IFT) [33] is a fine-grained approach to address security and reliability problems, including preventing sensitive information leakage [14], detecting SQL injections [41] and debugging [15, 24]. Security tags are attached to variables in a program, and these tags will propagate through computations. By propagating and checking tags (e.g., checking tag in I/O functions), fine-grained information control policies are enforced. DyTan [9] shows that IFT can be used to build a unified framework for various security problems. Moreover, various IFT systems have been built for mobile apps [14, 40], cloud services[35] and server programs [21].

However, no IFT system exists for big-data, and we attribute it to two major challenges. First, existing IFT systems incur high performance overhead, especially for data-intensive computations. We ran a recent IFT system Phosphor [3] in Spark with a WordCount algorithm on a dataset that is merely 200MB, and observed 128X longer computation time compared with the native Spark execution (§8.3).

The second challenge is on the architecture of DISC frameworks. DISC frameworks usually contain shuffle procedures which redistribute data across hosts (DISC frameworks' worker nodes). However, most existing IFT systems ignore dataflows across hosts. For the few [35] who support cross-host dataflows, transferring all tags in shuffles consumes excessive network bandwidth. Therefore, efficient cross-host tag propagation is crucial but missing in DISC.

This paper presents Kakute[1], the first precise and fine-grained information flow analysis system in DISC frameworks. Our key insight to address the IFT efficiency challenge is that multiple fields of a record often have the same tags. Leveraging this insight, we present two new techniques, Reference Propagation and Tag Sharing. Reference Propagation avoids unnecessary tag combinations by only keeping the *lineage of tags* in the same UDF, while Tag Sharing reduces memory usage by sharing tags among multiple fields in each record. To tackle the architecture challenge, Kakute completely captures dataflows in shuffles, and it efficiently reduces the amount of transferred IFT tags using Tag Sharing.

We implemented Kakute in Spark. We leverage Phosphor [3], an efficient IFT system working in the Java byte-code level. Kakute instruments computations of a Spark worker process to capture dataflow inside user-defined-functions (UDFs). Dataflow information is kept in a worker process and Kakute propagates it to other processes while shuffling. Therefore, IFT is completely captured across hosts and processes. In this paper, DISC frameworks and Kakute are trusted; UDFs are untrusted and they may be malicious or buggy. Kakute provides different granularities of tracking with two types of tags: `Integer` and `Object` tags. `Integer` provides 32 distinct tags, which is suitable for detecting information leakage and performance bugs. `Object` provides an arbitrary number of tags, which is suitable for data provenance and programming debugging. Kakute provides a unified API to tackle diverse problems. Based on this unified API, we implemented 4 built-in checkers for 4 security and reliability problems: sensitive information leakage, data provenance, programming and performance bugs.

We evaluated Kakute on seven diverse algorithms, including three text processing algorithms WordCount [38], WordGrep [25] and TwitterHot [38], two graph algorithms TentativeClosure [38] and ConnectComponent [38], and two medical analysis programs MedicalSort [36] and MedicalGroup [36]. These algorithms cover all big-data algorithms evaluated in two related papers [17, 19]. We evaluated these algorithms with real-world datasets that are comparable with related systems [8, 17, 19]. We compared Kakute with Titian [19], a popular provenance system, on precision and performance. Our evaluation shows that:

- Kakute is fast. Kakute had merely 32.3% overhead with `Integer` tag, suitable for production runs.
- Kakute is precise in data provenance. Compared with Titian [19], Kakute drilled down the actual bug-inducing input records by using `Object` tags, a huge reduction of 3 to 9 orders of magnitude reduction. Meanwhile, Kakute had 167% performance overhead on average, comparable with Titian and suitable for testing runs.
- Kakute effectively detected 13 real-world security and reliability bugs presented in other papers [11, 17, 37].

The main contribution of this paper is the first precise and unified IFT system for big-data. Other contributions include two new techniques that make IFT efficient and complete in DISC. Kakute can be applied to tackle a broad range of security and privacy problems in big-data (e.g., integrity verification [27] and privacy-preserving data mining [31]).

The remaining of the paper is organized as follows. §2 introduces the background of data-intensive scalable computing and information flow tracking. §3 gives an overview of the whole system architecture and workflow. §4 introduces dataflow tracking runtime in Kakute. §5 provides detailed information of our system design. §6 introduces our system API for building applications and how we use that api to resolve 4 reliability problems. §7 gives implementation details, §8 presents our evaluation results, §9 introduces related work, and §10 concludes.

2 BACKGROUND

2.1 Data-intensive Scalable Computing

DISC frameworks (e.g., Spark [43], DryadLINQ [42], MapReduce [12]) are popular for computations on tremendous amounts of data, to finish tasks like data analysis and machine learning. Computations are split across hosts and run in parallel, such that computation resources are efficiently used. Shuffles are frequent and sometimes are performance bottlenecks in DISC.

Many-to-many transformations (e.g., `groupByKey`, `join` and `aggregateByKey`) are prevalent in DISC. Each many-to-many transformation takes many input records and generates many output records. Given a buggy output record, extant data provenance system [8, 17, 19] will report all input records going through this transformation, including many irrelevant input records that generate other output records.

To avoid excessive computation, many DISC systems adopt the lazy transformation approach. [34, 42, 43]. Spark uses lazy transformations (e.g., `map`) for efficiency, and calls to these transformations

[1]Kakute is a precise, multi-purpose weapon used by Ninja.

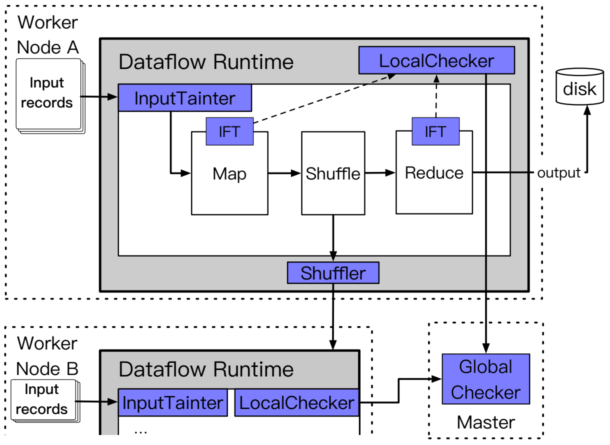


Figure 1: Kakute Architecture.

only create a new data structure called `RDD` with `lineage`. The real transformations are only triggered when collecting operations (e.g., `collect`, `count`) are called. These collecting operations trigger transformations along lineages, where unnecessary computations are avoided. Kakute leverages the lazy transformation feature in DISC to present its new Reference Propagation technique (§4.2).

2.2 Information Flow Tracking

Information Flow Tracking is initially proposed for preventing sensitive information leakage [33]. IFT attaches a tag to a variable (or object), and this tag will propagate throughout the computation. For example, a variable-level dataflow tracking will involve combinations of tags of two variables in each instruction, using an IOR operation. Different granularities of computation may incur different levels of computation overhead. Lower level (e.g., byte-level) tracking will consume a lot of resources, as each byte of data in an IFT system has its own tags [21].

Multiple research has been focusing on efficiency and applications of IFT. Shadowreplica [20] proposed to make use of the multicore resources while SHIFT [7] suggests accelerating dataflow tracking with hardware support. Several research [15, 24] adopts IFT for providing debugging primitives to improve software reliability. IFT has been applied to various areas, such as preventing sensitive information (e.g., GPS data and contacts) leakage in cellphone [14, 40], providing secure cloud services [35] and server program runtime [21]. To the best of our knowledge, no IFT system exists for big-data.

3 OVERVIEW

Kakute currently works with Spark, and it can be integrated with other DISC frameworks [34, 42] as long as these frameworks have UDFs, work with Java and adopt lazy transformation.

3.1 Threat Model

We consider a data processing service with multiple parties. The hardware infrastructures, DISC frameworks and Kakute are trusted. An organization owns some sensitive data, and the data is trusted. The organization allows a third-party analytic to write UDFs as programs to process data. The third-party analytic may cooperate with a malicious attacker that tries to get the sensitive information. Therefore, the programs running in the DISC frameworks are not trusted, and they may leak data through UDFs or output.

3.2 Architecture

Spark has a master scheduling tasks and collecting final results, and workers conducting the actual computation. Kakute adds 5 components in Spark to do IFT transparently, so that people who write UDFs do not need to be aware of Kakute.

Figure 1 shows the architecture of Kakute with 5 key components: DataflowRuntime, InputTainter, LocalChecker, GlobalChecker and Shuffler. We leverage Phosphor (§4.1), a recent IFT implementation, with variable-level IFT by instrumenting Java bytecode.

In Kakute, input records go through InputTainter which automatically adds tags to the inputs. Throughout the whole computation procedure, tags of data will be propagated within a worker and across hosts. In each DISC transformation, Kakute users can decide if tags of the computation results will be sent to GlobalChecker. Finally, Kakute users can decide whether tags going to network and filesystem should be checked by LocalChecker. Kakute users implement their decisions using checkers (§6).

DataflowRuntime provides the basic dataflow tracking functionalities in Kakute. It intercepts Java `ClassLoader` and instruments Java bytecode to provide tag storage and propagation code. Cross-host propagation (§4.4) is implemented in this component.

InputTainter automatically attaches tags to the inputs of applications. InputTainter intercepts input functions (e.g., `textfile` in Spark), and uses the APIs provided in Table 1 to automatically add tags to input records. Programmers can write their own InputTainter by specifying taint policies (§5.2).

LocalChecker enforces per-worker security policies (e.g., a sensitive field must not flow to an I/O function in a UDF). Kakute allows data providers to specify sensitive fields in data records (§6.2), and it allows security experts who write the LocalChecker to specify dangerous functions (§5.3). On a worker node, a LocalChecker intercepts each UDF with an IFT module.

GlobalChecker enforces cross-host policies (e.g., aggregating the number of suspicious I/O events among all hosts). Kakute allows security experts to collect reports from the LocalChecker in each host (§5.4).

Shuffler intercepts shuffle flows to efficiently carry IFT tags across hosts (§4.4).

3.3 Example

We consider an example used in data provenance system [17, 19]. Suppose a data scientist in a movie live-stream website writes a program to find out the top-5 ranked movies of each user from a user-movie rating database. The input of the program is a collection of rating records in the form of ⟨userId, movieId, rating⟩, describing a user's rating for a movie. The program first uses a `groupByKey` operation to group rating records for each user, then sorts records in each group to find out the top-5 rated movies. However, this data scientist is not well trained and her program crashes on the (`John`, `MovieC`, 5) output record.

Data provenance with high precision is crucial for identifying bug-inducing input records. Figure 2 shows the precision of Kakute

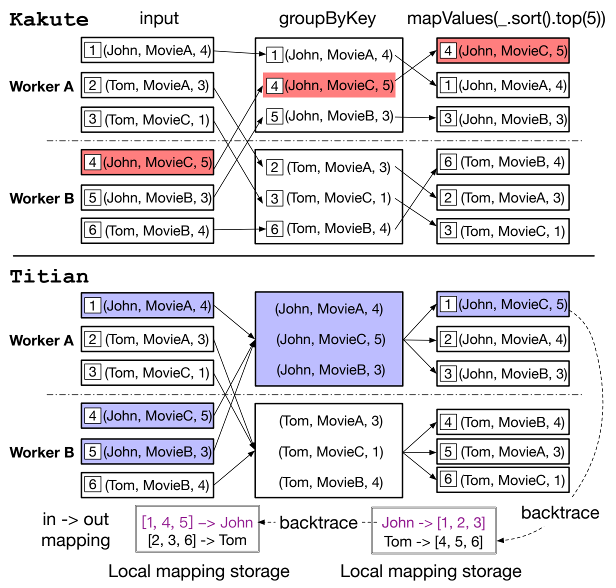


Figure 2: Precision of Kakute and Titian on tracing back the (John, MovieC, 5) record. A solid rectangle represents a record. Red records are Kakute's back-tracing records and blue records are Titian's.

and Titian: Kakute drills down the only one bug-inducing input record, while Titian reports all input records going through the groupByKey transformation that produces (John, MovieC, 5). Kakute has much higher precision than Titian because Kakute assigns an input record with a unique ID "4" using Object tag, and this ID propagates through transformations. Tracing back inputs from an output record can be achieved by directly retrieving IDs in the output record with Kakute's API (§5.1).

4 EFFICIENT IFT

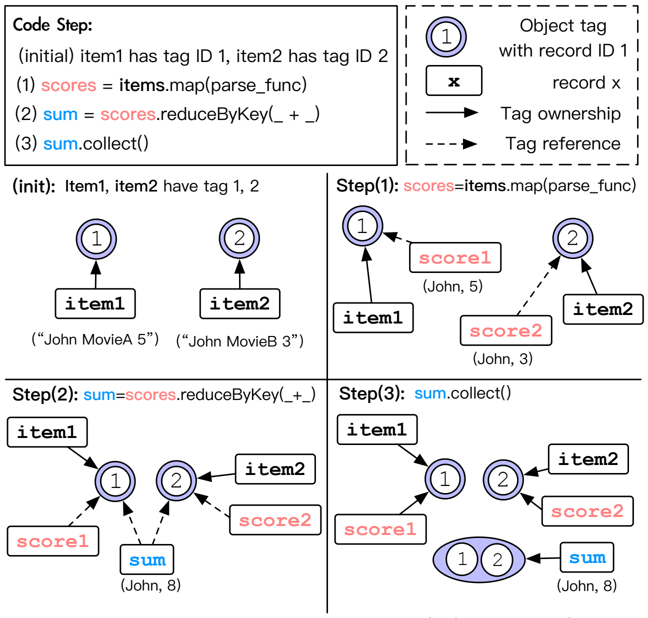
In the section, we introduce an efficient and fine-grained IFT system for DISC frameworks. We first introduce Phosphor and then present our techniques to make IFT efficient and complete in DISC frameworks.

4.1 Phosphor Background

Phosphor [3] is a recent portable and precise IFT system in Java. Phosphor instruments Java bytecode, by adding tag fields to class definitions and local variables in functions, and adds propagation code to function bytecode. Phosphor adds shadow fields for primitive variables (e.g., int and double) and an extra field for an object to store the tag. As native code execution exceeds scope of JVM, Phosphor can not precisely propagate tags in native methods. Therefore, Phosphor adopts the approach of combining all tags in function arguments and propagates them to all return variables.

Phosphor provides Integer and Object tags. For Integer tags, each tag is a 32-bit Integer initialized to 0, so at most 32 distinct tags are available. For all operations, Phosphor directly does a IOR (a bit-wise OR) on tags of all operands.

Phosphor handles Object tags in a different way. For assigning operations in Java, Object tags are copied. Each Object tag (initialized to null) has a list of tag labels. For arithmetic operations such as + and -, an new Object tag is created for combining Object



Code Step:
(initial) item1 has tag ID 1, item2 has tag ID 2
(1) scores = items.map(parse_func)
(2) sum = scores.reduceByKey(_ + _)
(3) sum.collect()
Object tag with record ID 1
record x
Tag ownership
Tag reference
(init): Item1, item2 have tag 1, 2
item1
item2
("John MovieA 5")
("John MovieB 3")
Step(1): scores=items.map(parse_func)
score1
(John, 5)
score2
(John, 3)
Step(2): sum=scores.reduceByKey(_+_)
sum
(John, 8)
Step(3): sum.collect()

Figure 3: Tag Propogation process with the given code.

tags from operands of the operations. Details of the propagation rules can be found in a previous work [3].

To improve Phosphor efficiency, Kakute instruments only the Java bytecode of UDFs. The byte-code instrument procedures are done at runtime. When a Java class file is loading into JVM, the intercepting agent will instrument its byte-code and stores the instrumented code into a local cache file for latter usage. Kakute just reads the cached instrumented byte-code when the class is loaded next time, avoiding excessive instrumentations.

4.2 Reference Propagation

Kakute extends the original tagging system with Integer and Object tags. Integer tags have better performance in most cases while the number of distinct tags is unlimited for Object. Object tags can support an unlimited number of tags, bringing more flexibility for IFT. We keep the original propagation design of Phosphor for Integer tag, which simply does an IOR for tags of two operands that take part in a computation.

For Object tag, however, we reconstruct the whole tagging system. Our evaluation shows that Phosphor's list-based tagging design is not suitable for DISC frameworks, because a tremendous number of tags exist in these frameworks and combinations of two tags are extremely time-consuming (§8.3). The cost of combining two tags is proportional to the length of the tag lists in these two tags.

We present a new technique called Reference Propagation for making Object tag propagation efficient in DISC. This technique leverages Spark's lazy transformation feature (§2.1). Spark firstly stores the transformation sequences of records as lineage, it then executes the lineage only when collecting operations (e.g., collect) are called. Similarly, Kakute tracks the lineage of the Object tags by using references of these Java objects, and it only computes the actual Object tags on collecting operations.

Figure 3 shows the idea. Initially, two string records, item1 ("John MovieA 5") and item2 ("John MovieB 3"), are input records, and

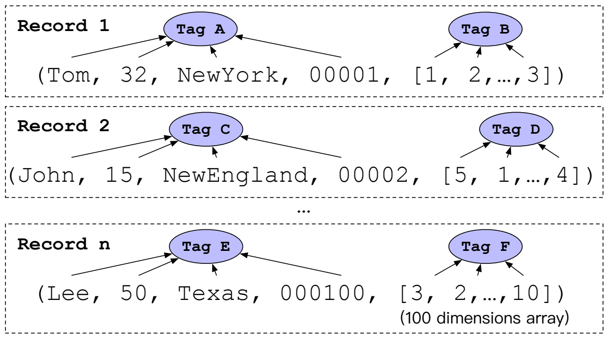

Record 1
Tag A
Tag B
(Tom, 32, NewYork, 00001, [1, 2,…,3])
Record 2
Tag C
Tag D
(John, 15, NewEngland, 00002, [5, 1,…,4])
...
Record n
Tag E
Tag F
(Lee, 50, Texas, 000100, [3, 2,…,10])
(100 dimensions array)

Figure 4: Tag object sharing between fields in each record.

they have `Object` tag ID 1 and 2 respectively. After Step (1), the first and third parts of each rating string are parsed as a tuple, and record `score1` and `score2` are generated from `item1` and `item2`. The reference of record `item1`'s tag will propagate to `score1`. As a result, no extra tags are created. After Step (2), record `score1` and `score2` have the same key and are combined to a new record `sum`. At this moment, `sum`'s tag is still a reference to two existing tags. After Step (3), a new tag for `sum` is generated, and its labels is the combination of two existing tags of `score1` and `score2` (based on the Spark lineage between `scores` and `sum`).

Overall, Phosphor takes four tag computations (arrows in dash) in total, including two in Step (1) and two in Step (2), because Phosphor's tag propagation is not lazy. In contrast, KAKUTE has only two tag computations in Step (3), as it only propagates references of tags in Step (1) and Step (2). Reference Propagation greatly reduces unnecessary tag propagation (§8.3).

4.3 Tag Sharing

We observed that different fields of a data record often share the same IFT tags in DISC frameworks. KAKUTE introduces a Tag Sharing mechanism. Upon adding tags to fields, KAKUTE first checks if this tag is in its own per-host cache, and it retrieves the corresponding `Object` tag directly if it exists. If there are no such tags, KAKUTE creates a new `Object` tag and keeps it in the cache. Tag Sharing adopts a simple FIFO cache mechanism. KAKUTE computes the hash of an `Object` tag and determines if there is a cache hit or not.

Figure 4 shows how different fields of a medical record share tag objects in KAKUTE. Each record has name, age, city, ID and a list of numbers that represents one's medical examination results. In each record, patients' personal information can have the same high sensitivity tags and all examination results have the same low sensitivity tag. Therefore, KAKUTE maintains only two tags for each medical record, as shown in Figure 4, greatly reducing memory space and computation overhead compared to conventional IFT systems (§8.3).

4.4 Tag Propagation Across Hosts

In KAKUTE, both records and tags are modeled as Spark tuples. For instance, if a data record is (1, (2, 3)), then the tag tuple can be (tagA, (tagB, tagB)). Adding, removing and getting tags can be done by KAKUTE's coarse-grained APIs (Table 1) or InputTainter (§5.2).

To make IFT complete in DISC frameworks, KAKUTE is able to propagate tags across hosts and to persistent storages like disk

Function	Description
Coased-grain API	
setTaint(T → Any) : RDD[T] → RDD[T]	set tuple tags for each record in a data collection
removeTaint() : RDD[T] → RDD[T]	remove all tags for each record
getTaint() : RDD[T] → RDD[(T, Any)]	get data tuples along with its tags, to ensemble (data, tag) pairs
InputTainter and TaintChecker	
setChecker(IFTChecker)	set the checker to specify tracking policies, inputTainter and so on.
Appllication API	
conf.dft.traffic.profiling, conf.dft.traffic.scheme	start profiling of application performance; choose partition scheme from profiling
conf.dft.security=confFile	set up fine-grained information control
traceBack(record) → inputRecordSet	trace back to input of producing these records
getErrorRecords → errorRecords	return records that throw exceptions
getErrorRecordsWithUDF → List[(errorRecord, func)]	return records and UDF that throw exception
coverageAdd(RDD[T])	add the coverage dataset, so that the coverage checker will track the dataset
coverageTest(out, RDD[T])	test coverage of the input dataset

Table 1: KAKUTE API

when shuffling or checkpoint happen. As both data and tags are tuples, KAKUTE adopts a straightforward approach of wrapping tags and data into (data, tag) pair and transfers this pair to other hosts or disks. When other hosts get this pair, they unwrap it and attach the tags to the data.

When wrapping the (data, tag) pair, extra computations may be necessary. With Integer tags, KAKUTE directly retrieves tags from data and generates the (data, tag) pair. On the other hand, with `Object` tags, tags are computed from the lineage in a `Taint`, and tags are represented by an array of tag labels. After tags are computed, the lineage of a `Taint` is cut (dependency will be set to `null`), so that unused `Taint` objects will be collected by Garbage Collector (GC).

KAKUTE further adopts a key-value representation of tag tuple to reduce network bandwidth. For example, a tag tuple (tagA, (tagB, tagB)) can be translated to Map(1 → tagA, 2 → tagB , 3 → tagB). Then, when tags are 0 or null, this Map can be compressed. In the previous case, if tagA, tagB are both `null`, the key-value representation is `Map()`. Therefore, shuffle network bandwidth is greatly reduced when only some data in the system have tags.

KAKUTE also provides fine-grained supports for objects, array, Map and iterable objects. when generating the tag tuple, KAKUTE retrieves tags from each element of an iterable object, and put them to an `ArrayTag` which models a list of tags in an iterable object. For support of user-defined objects (e.g., Student), KAKUTE retrieves instrumented fields in `ObjectTainter` (§5.1) and propagates only those tags to other hosts or disks. For objects without `ObjectTainter`, KAKUTE get tags from all their fields, and combines them into a single tag, then this single tag will propagate to other hosts or persistent storage. At the time when other hosts receive this tag, KAKUTE attaches this tag to this object itself and all its fields.

5 KAKUTE RUNTIME

This section introduces KAKUTE's runtime. We first explain usage of KAKUTE's APIs and then give the design of each runtime component.

5.1 Kakute API

Table 1 shows Kakute's APIs for manipulating tags (e.g., `set` and `get`). Adding tags to a collection of data is straightforward in Kakute. Figure 5 illustrates how to use Kakute's API to set or get tags for a shopping order dataset (each record contains time, creditCardId, productId).

Kakute supports fine-grained tagging for objects (e.g., user-defined Student). To add tags to a user-defined class, programmers need to implement an `ObjectTainter`, which tells the framework the fields in an object that should be tagged. In fact, `ObjectTainter` is a wrapper of a list of field names in a class. Kakute will read these field names and add tags to the fields of a object through Reflection in Java. If no `ObjectTainter` is provided for a object, tags will be added to all fields of the object.

For tag retrieval, programmers can use `getTaint`() to get data along with its tag as (data, tag) pair. With `getTaint`, programmers are able to get tags of a specific data record. Other APIs including `removeTaint`, which will get rid of tags in specific records. Full list of APIs is in Table 1.

```
// add tags 0, 1, 0 to time, creditCardId and productId
order_tag = order.setTaint(t => (0, 1, 0))

// add tags only when the order use credit card
order.setTaint(t => if (t.creditCardId != 0) (0, 1, 0) else 0)

// return tag of orders as (order, tag) pair
orderWithTag = order_tag.getTaint()
```

Figure 5: Code for manually adding/getting tags of data.

5.2 InputTainter

`InputTainter` is used to provide configurable APIs for automatically adding tags to input data. In most information flow analysis programs, only input data needs to be tagged. Programmer can specify their own implementation of InputTainter which takes input type and input detailed information (e.g., filename) as function parameters and it returns the tagged input. InputTainter intercepts input data functions (e.g., `textfile` and `parallel` in Spark), so each data record is tagged. We implemented a SringInputTainter that provides char-level tagging for string input. It first reads a metadata file (see the format in Figure 6), splits the string, add tags according to the meta file and combines into the original string.

5.3 LocalChecker

LocalChecker is to provide a user-defined interface for checking tags in some particular points (e.g., I/O functions). Programmers can provide a checker function in `LocalChecker` for some dangerous functions. For example, `writeInt(int)` → `checkFunc` means that `checkFunc` will be invoked before `writeInt` is invoked. For information flow tracking, a instrumented `writeInt(int)` is modified to `writeInt(int, Taint[])`, and the extra parameters (of type `Taint`) are tags of the original parameters. All tags of the intercepted functions will be passed to the `checkFunc` which is defined as `checkFunc(Taint)`. In information control (Figure 6), checkFunc checks if there are security tags of the function parameters, notifies GlobalChecker and terminates the program if the tag is not 0 (sensitive). This instrument process is done by the intercepting agent that

```
input.txt.meta
// only the second column of input records (creditCardId) is tagged
SecurityLevel=0, 1, 0
```

```
PrivacyLeakageChecker.scala
class PrivacyLeakageChecker extends IFTChecker {
  override val inputTainter = (in: Iterator[Any], file: String) => {
    val levels = parse(file + ".meta")
    setTaint(in, levels)
  }
  // tags of parameters in intercepted functions (I/O function)
  // are passed to checkerFunc (tag of i in function writeInt(int i))
  override val localCheckerFunc = (tag: Int) => {
    if (tag != 0) {
      notifyGlobalChecker(IllegalAccess(getStackTrace))
      throw new IllegalAccessError("output contains tags")
    }
  }
  override val globalChecker = (msg: Message) => {
    case e:IlleagalAccess => // write e to log
  }
}
```

Figure 6: Meta file format and the checker for information control. Tags are automatically attached to input records by InputTainter.

instruments Java byte-code. The `LocalChecker` can also collect other information such as exceptions to the `GlobalChecker`.

5.4 GlobalChecker

GlobalChecker is for cross-host information collections in Kakute. GlobalChecker is a component in master that collects information from the LocalChecker in every worker. On a particular event of UDF execution, data shuffling and exceptions, per-host event information is sent from all LocalCheckers to the GlobalChecker. For example, the GlobalChecker of a information control checker gets all violations of security policies from local checkers (Figure 6).

6 KAKUTE CHECKERS

This section first introduces a guideline for developing checkers, then presents checkers for 4 security and reliability problems, including fine-grained information control, data provenance, programming bugs and performance bugs.

6.1 General Pattern of Developing Checkers

To develop a checker, programmers need to implement 3 elements: `InputTainter`, `LocalChecker` and `GlobalChecker`. First, an `InputTainter` is to specify how input records should be tagged. Second, the role of `LocalChecker` is intercepting specific functions, and return result to `GlobalChecker`. Third, the `GlobalChecker` defines the behaviors after receiving checking results from LocalChecker. In total, each checker took only 52 to 101 LoC to implement in Kakute.

6.2 Fine-grained Information Control

IFT is used for providing fine-grained information control to prevent leakage of sensitive information for years [14, 33]. Each variable is attached with a tag and propagated throughout all computation. And information control policies are enforced in some functions (e.g., I/O and system call). We built fine-grained information control in big-data semantics, preventing information leakage through computation or result output in DISC frameworks.

We adopted a similar tag design as TaintDroid [14]. 32 distinct tags are used to define different security levels which are enough for most programs [40]. When two variables are involed in an operation, tags are also combined using an IOR operation. We implemented an InputTainter that read an extra file that contains security levels for every fields in inputs. If input file is `in.txt`, then the tag configuration file should be `in.txt.tag`. We implemented a LocalChecker that intercepts every I/O functions, and it checks whether data through I/O functions has security tags. If tags are found, a message with details of tags and data will be sent to the GlobalChecker. LocalChecker also checks if there are tags in output functions (e.g., `collect`). We made a GlobalChecker that asks the LocalChecker to stop workers as long as there are security tags in I/O or output functions.

6.3 Data Provenance

Data provenance is crucial for many security problems, including post-incident investigation [2], integrity verification [27] and privacy control [26] There are two kinds of tasks in data provenance: finding inputs that produce specific output records (backtracing) and finding outputs that specific input records will produce (forward-tracing). Back-tracing is for finding inputs that produce specific outputs, while forward-tracing is for finding outputs that some input records produce. Precision is crucial for both forward and backward tracing. For example, extra efforts should be make to identify the actual input records in a back-tracing if the number of back-tracing records is too many. We wrote an InputTainter that adds unique tags to all records in a data collection. These tags are unique to identify input records, and will propagate through computations. To trace an output record to its input, programmers retrieve tags from the record and get the input records accordingly. We implemented a `trace` function (Table 1), and it automatically retrieves tags from the traced records and returns the related input records.

Kakute's approach for data provenance is precise in DISC frameworks. Many-to-many operators with UDFs in DISC frameworks make a black box of transformation, making it difficult to get precise tracing result. Traditional data provenance systems also have similar problems [10]. However, IFT opens the black box of transformation. Propagation of the tag with IFT is fine-grained and more precise data provenance support is achieved.

6.4 Programming Bugs

Debugging programs is tricky and time-consuming [17] in DISC frameworks. A bug may be caused by multiple transformations with UDFs. We built a debugger that preserves fine-grained lineage information, which helps programmer to identify and fix bugs easily.

Kakute provides the following debugging methods:

Record Backtracing returns the input records that produce an exception or problematic outputs. After running the program again with these few input records, programmers identify the problems in a step by step debugging (with JDB).

Local Replay collects the UDF and inputs when there are errors or exceptions. After running the program locally with these input, programmers may find out the programming bugs.

Flow Verification verifies program flows defined by programmers. For example, when multiplying Matrix A_{2*2} and Matrix B_{2*2}, the result Matrix $R_{1,1}$ must be computed from $A_{1,1}$, $A_{1,2}$, $B_{1,1}$ and $B_{2,1}$. Programmers can provide this rule for verifying the MatrixMultiplication programs. The debugger uses some default rules for verifications: all input data should be covered in the computation result, and a field in the output should not come from nowhere. The debugger will show warnings when there are such problems.

We wrote an InputTainter that attaches unique tags for each field in records. Each record is identified by an id, and fields in a record are identified by an index. Tags are in the form of (recordId, fieldId) for each field. For instance, (John, 32, Ken) can have tag ((1, 1), (1, 2), (1, 3)). We implemented a LocalChecker that collects error records and their UDFs. This LocalChecker sends exception information to GlobalChecker. GlobalChecker gets these records and traces these records to input. Then, Record Backtracing and Local Replay are achieved. GlobalChecker also translates flow rules like (out entry 1 → in entry 2) to rules of checking tag 2 in output.

6.5 Performance Bugs

Shuffle operations in DISC frameworks can consume excessive I/O resources with naive partition of data. An efficient partition scheme can greatly reduce shuffle traffic and speed up computation by partitioning only some fields of records [45]. Consider this transformation: `rating.map(t→(t._1._1, t._2)).reduceByKey(_+_)`. One record ((John, MovieC), 5) in `rating` has the transformation of ((John, MovieC), 5) → (John, 5) → (John, 8). Shuffles in `reduceByKey` will partition data on "John". If an efficient partition scheme that only partitions data records based on the user name field can be inferred automatically, then shuffle traffic will be 0 compared with a random partition scheme on all fields.

We built a checker for automatically inferring an efficient partition scheme. We implemented an InputTainter that adds unique tags to each field in a record, upon each UDF input. Then, we implemented a LocalChecker to collect tags from each field in records of each UDF output. Thus, the field dependencies of each transformation are collected by simply retrieving tags from each field. The GlobalChecker collects this information from all LocalChecker and combines them together. With the inferred field dependency of UDFs, we adopted a simple backtrace algorithm that computes the efficient partition scheme (similar to [5, 45]). Kakute's partition scheme is inferred in profile run with small set of data, so Kakute will not bring any overhead to production runs, making its inferred partition scheme a free gift to improve production run performance.

7 IMPLEMENTATION

We implemented Kakute in Spark [43] using Scala with 3500 LoC. We also modified Phosphor for our Reference Propagation and Tag Sharing with several bugs fixed. Most of our implementation is independent of Spark except for the performance bug checker, because this checker needs to modify Spark in order to get UDF source code debug information (§6.5).

7.1 Adding, Getting and Removing tags

We wrapped APIs provided by Phosphor to provide general APIs for tag manipulation. To provide the high-level APIs in §5.1, we extend 3 types of RDD objects in Spark, namely TaintedRDD, WithTaintRDD and TaintLabelRDD.

A `TupleTainter` uses pattern matching to match multiple fields in a data record with tag objects. Tags matching can be many-to-many or many-to-one. One tag object might be referenced by many fields of a data record. In this case, only one tag object tracking multiple fields is transferred to other hosts during cross-host propagations. This decreases much propagation traffic across hosts.

A `ObjectTainter` adds tags to a general object. It first reads field names and use `Reflection` in Java to set taints for fields contained in the object. If no `ObjectTainter` is defined for the object, the whole object is matched with one taint object.

7.2 Non-synchronization Tag Cache

In previous section (§4.3), we show how Tag Sharing reduces tag numbers in the system. With Tag Sharing, KAKUTE tries to avoid redundant tags as much as possible. KAKUTE avoids the cost of synchronization and use `ThreadLocal` for Tag Sharing. Each cache of tag is isolated among threads, and incurs no synchronization. KAKUTE will create a array of cache that store `Taint`, and this cache will be updated if a new tag comes with duplicated hashcode.

7.3 Handling Implicit Information Flow

KAKUTE tracks data flows accurately, but it may lose implicit data flows. In a WordCount program, if a record ⟨`word`⟩ is transformed to ⟨`word, 1`⟩, the field `1` contains no tag. We handle this problem by developing a inheritance approach as below. When a new field is generated from tagged parent records, the field is attached with tags from parent fields. This conforms to our observations that newly generated fields inherit information from parent fields in most DISC algorithms.

7.4 Fault Tolerance

KAKUTE's implementation only modifies a worker's computation code, so fault tolerance property of Spark is preserved. In spark, failed tasks will be submitted and computed again according to their lineage. In KAKUTE, input data is tagged automatically by InputTainter, and tags are checkpointed along with data, so tags can be recomputed when data computation is done again. Therefore, tag computations are fault-tolerant in KAKUTE.

Collector and Checker should also be able to handle failure in KAKUTE. When a worker process fails, the Collector will lose communications with the master. Then, a new worker will be created and the checker can work again. When a task fails with the worker working properly, the Collector will not collect Checker information for this task, and then sends a failure message to the master. Therefore, collected information in the failed tasks will not affect the final IFT analysis result.

8 EVALUATION

We evaluated KAKUTE on its (1) tracking precision (2) computation overhead, (3) bandwidth overhead (4) vulnerabilities detection. We showed that KAKUTE has a low overhead while keeping the fine-grained information flow tracking property. We used a cluster with 9 machines, each is equipped with Intel Xeon 2.6GHz CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD.

We evaluated seven programs on KAKUTE and Titian for their precision and general runtime overhead. We chose Titian as it is the only open-source data provenance system for big-data. These programs include three text processing algorithms WordCount, WordGrep and TwitterHot, two graph algorithms TentativeClosure and PageRank, and two medical analysis programs MedicalSort and MedicalGroup. These algorithms cover all big-data algorithms evaluated in two related papers [17, 19]. We evaluated all these seven algorithms with several real-world dataset, which are comparable with existing systems [8, 17]. Datasets and programs are showed in Table 2.

App Name	Dataset	Size/Records
ConnectedComponent	TwitterSocial [23]	1.5B edges
TentativeClosure	TwitterSocial [23]	1.5B edges
MedicalGroupBy	MedicalDB	100M records
MedicalGroupSort	MedicalDB	100M records
TwitterHot	TwitterStream	50GB
WordCount	Wikipedia	100GB
WordGrep	Wikipedia	300GB

Table 2: Evaluation applications and datasets. All datasets are comparable with previous research.

Our evaluation focuses on these questions:

§8.1: What is KAKUTE's precision compared with Titian?
§8.2: What is KAKUTE's performance overhead ?
§8.3: How can Reference Propagation and Tag Sharing reduce runtime overhead of KAKUTE?
§8.4: Is KAKUTE scalable to large datasets?
§8.5: Can KAKUTE effectively detect security and reliability bugs?

8.1 KAKUTE v.s. Titian

In order to evaluate KAKUTE's precision in terms of dataflow tracking, we evaluated KAKUTE on seven algorithms (or programs). These seven applications are TwitterHot, ConnectedComponent, MedicalGroup, MedicalSort, TentativeClosure, WordCount and WordGrep.
TwitterHot finds popular words in a period of time. It puts all Twitter posts into different groups with different time, and then counts the words in each group. The time interval for each group is 1 minute. The algorithm first uses `groupByKey` to divide data into groups and counts words in each group.
ConnectedComponent is to compute connected component in a graph. We used the label propagation algorithm that assigns a label to each node, and each node exchange their labels with all neighbours and only keeps the smallest label. In practice, `join` is used for exchanging ones' label with neighbours. `union` and `reduceByKey` are used for combining labels.
MedicalGroupBy samples patients according to their age. Patients are divided into five groups, and then, 50 patients will be chosen randomly from each group. A program that uses `groupByKey` and `filter` will solve the problem. **MedicalSort**, on the other hand, will find top-10 patients with the most serious disease in each group.
TentativeClosure computes a collection of nodes that one node can access. Each node maintains a list of accessible nodes, which is initialized as itself. In each iteration, a node will `join` the accessible list and all edges to get the new accessible list. Repeating this procedure, until the number of accessible nodes of a node is not changing anymore.

Table 3 and Table 4 show backward and forward tracing results of KAKUTE and Titian, the state-of-art data provenance system of Spark. Compared with Titian, KAKUTE shows much better (at

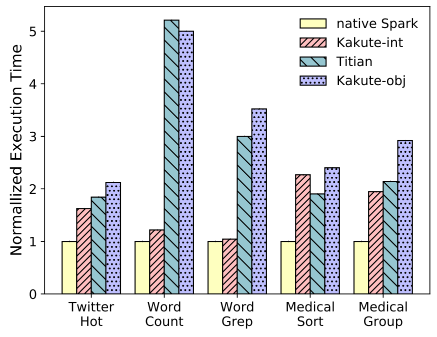


Figure 7: Overhead of Kakute and Titian

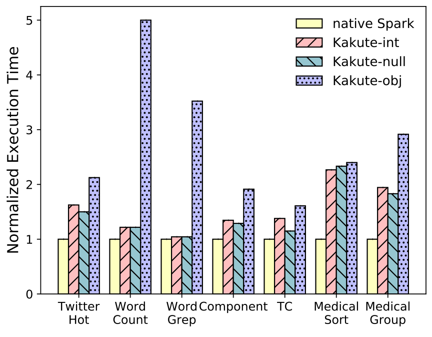


Figure 8: Kakute Computation Breakdown.

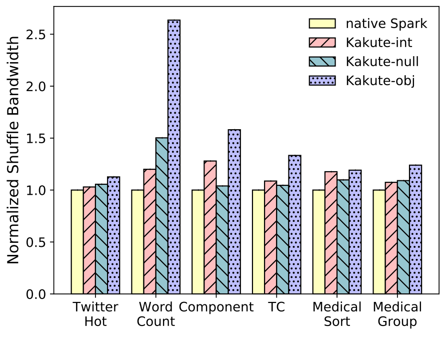


Figure 9: Kakute Shuffle Bandwidth Breakdown.

App Name	Traced Record	Titian	Kakute
TwitterHot	"strike"	2.75×10^5	520
ConnectedComponent	random	1.4×10^9	4
MedicalGroupBy	one elderly	1.99×10^7	1
MedicalGroupSort	one patient	2.56×10^7	1
TentativeClosure	random	1.31×10^3	1
WordCount	"religion"	6.03×10^4	6.03×10^4
WordGrep	"science"	1.37×10^5	1.37×10^5

Table 3: Kakute's backward tracing result.

App Name	Traced Record	Titian	Kakute
TwitterHot	"#FAKENEWS"	1.50×10^6	1609
ConnectedComponent	random node	1.58×10^6	126
MedicalGroupBy	one elderly	2.0×10^7	1
MedicalGroupSort	one patient	1.41×10^7	1
TentativeClosure	random node	1.31×10^3	1
WordCount	"religion"	6.03×10^4	6.03×10^4
WordGrep	"science"	1.37×10^5	1.37×10^5

Table 4: Kakute's forward tracing result.

System Name	iteration				
	1	**2**	**3**	**4**	**5**
Titian	69	1.3×10^6	2.9×10^8	1.1×10^9	1.4×10^9
Kakute	1	2	2	3	4

Table 5: Kakute's backward tracking with different iterations (ConnectedComponent).

most 7-9 orders of magnitude) precision for five algorithms except WordCount and WordGrep, because these five algorithms contain many-to-many transformations (e.g., `groupByKey`, `join`). Therefore, back-tracing or forward-tracing will return all input/output records that go through a transformation.

We observed that Titian gets more imprecise for iterative algorithms. Table 5 shows the backward tracing results of running 5 iterations with ConnectedComponent. The length of label propagation path of a connected component increase by 1 per iteration. Back-tracing result for Kakute precisely finds which edge propagates the label and hence increases by less than 1 per iteration. In contrast, Titian's back-tracing result grows exponentially. Titian considers all edges connected with the traced node as sources of label, due to the imprecise tracking level on many-to-one operation `reduceByKey` in label combination of ConnectedComponent. Titian returns almost all input records (1.4B) at 5 iterations.

8.2 Performance Overhead

Figure 7 shows Kakute's performance overhead compared with Titian on computing all output records and tracing back from these output records. We use the same dataset as in the prior subsection. "Kakute-int" denotes Kakute with `Integer` tags. "Kakute-obj" denotes running Kakute with an `Object` tag for each input record. Titian stores mapping of each operation, so costs of back-tracing are proportional to the computation time and dataset size, as it will need `join` on a large dataset. Kakute shows similar overhead compared with Titian, while Kakute has a much higher precision of tracing.

Figure 8 shows Kakute's runtime computation overhead. To break down Kakute's overhead, "Kakute-null" denotes running Kakute using `Object` tag with all tags `null`. Kakute has an overhead of 32.3% on average when using `Integer` tags. "Kakute-int" and "Kakute-null" show similar performance. For "Kakute-obj", WordCount suffers from a higher overhead of 425% , as its lineage of WordCount were long, causing the long tracing-back time.

Figure 9 shows Kakute's network overhead. We ran the 7 algorithms and save the final result with their tags using `getTaint`. For most of the algorithms, an output record may only come from some portions of input records, so the number of tags in shuffle is negligible, comparing with the original data. For WordCount, a frequent word (e.g., the "the" word) can be computed from a large number of input records, causing a large set of tags in shuffle.

8.3 Effectiveness of Reference Propagation and Tag Sharing

Table 6 shows performance improvement with Reference Propagation and Tag Sharing. We ran seven programs with smaller datasets, because the standard datasets used in previous subsections caused a long execution time in Phosphor for most programs. We compared computation time with 3 configurations: Phosphor, Reference Propagation only and Kakute (Reference Propagation and Tag Sharing). With Reference Propagation, execution time decrease by about 20% for WordGrep, TC, MedicalSort and MedicalGroup

Program	Phosphor	Reference Propagation only	Kakute
WordCount	inf	398s	297s
WordGrep	15min	12.8min	8.1min
Component	inf	inf	1992s
TC	80s	60s	51s
MedicalSort	45s	43s	42s
MedicalGroup	27s	18s	14s
TwitterHot	inf	inf	17s

Table 6: Computation time with two optimization techniques. "inf" means computation time is longer than 1h or throwing exceptions.

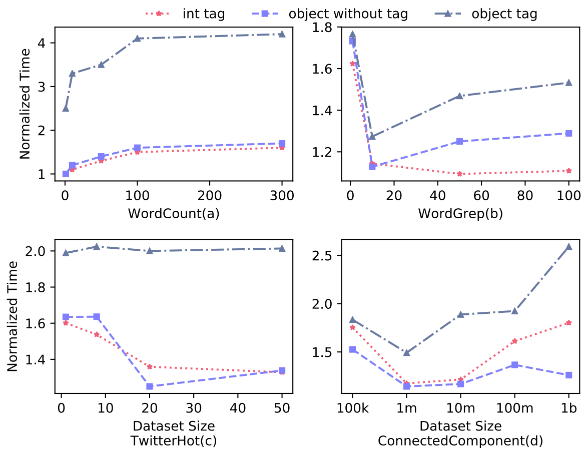


Figure 10: KAKUTE's scalability to datasets size.

Program	Native Spark	Naive Wrapping	KAKUTE
Component	32.9GB	42.1GB	34.2G
WordCount	2.6G	3.4G	2.9G

Table 7: Shuffle Traffic reduction of Key-Value Compression

Three other programs (e.g., WordCount) can not even finish execution in Phosphor. Reference Propagation only calculate tags upon collection (§4.2), so it greatly reduce computation time for WordCount from infinite to 398s. We found that running WordCount with Phosphor had 20.6 billion tag combining operations and most of its time was spent in combing these tags. WordCount running with KAKUTE had only 9.2 million combining operations. However, with Reference Propagation only, the Component and TwitterHot programs still had infinite execution time in KAKUTE due to the enormous amount of tags. TwitterHot running with Phosphor had a `GCOverheadLimitExceeded` Exception because the program had too many tags, while KAKUTE's Tag Sharing greatly reduced memory usage and brought TwitterHot's execution time down to 17s.

With both Reference Propagation and Tag Sharing enabled, all seven programs were able to finish within about 30min. For the MedicalSort program, KAKUTE and Phosphor had similar performance because the number of tags in this program is small. Table 7 explains why KAKUTE is much faster than Phosphor because its cross-host tag compression (§4.4) save much network bandwidth compared with uncompress (naive) tag propagation.

8.4 Scalability of Computation Overhead

Figure 10 shows that KAKUTE's computation overhead with datasets with different size in 4 programs. Among all applications, KAKUTE's computation overhead is 1X with Object Tags, while int and object with null tag have smaller overhead ranging from 50% to 1X. Overhead of these 4 applications increases slowly with increasing datasets size, because the number of input records that produce an output record is few. We observed that some algorithms have an overhead drop while increasing dataset size. The reason is that small dataset take a short time for executions, and starting a new executor might be slowed for the extra code instruments time. Therefore, this cost can be amortized, with increasing computation time.

8.5 Detecting Security and Reliability Bugs

This subsection evaluated 13 programs with security or reliability bugs presented in 3 papers [11, 17, 37]. Similar to existing IFT systems [3, 9], KAKUTE did not extensively look for new bugs. All 13 programs were written by us as they are not open-source.

8.5.1 *Fine-grained Information Control.*

We evaluated our fine-grained Information Control system with 2 programs.

Leakage Through I/O channel An attacker may submit a task that writes records with sensitive information (e.g., creditCardId) to disks or remote hosts. With KAKUTE, each output stream is checked to ensure that there are no security tags. We evaluated a malicious program MedicalAnalysis which is similar to a previous work [37]. It writes sensitive information to disk and network in a `map`. In evaluation, patients' personal information was tagged with security tags. When we ran this malicious program, a notification of I/O violation was sent to users and the worker terminateed. We also wrote non-sensitive information in this UDF to network, yet these operations were allowed as it did not contain any tags.

Leakage Through Output An attacker may submit a task that directly or indirectly outputs records with sensitive data. KAKUTE checks output of each task to ensure that every output records do not have any security tags. We evaluated a `WalmartAnalysis` program [1] that leaks some specific records. The input were shopping records of customers, and customer names had security tags. Our checker showed warning and forbid the output when running the program.

8.5.2 *Data Provenance.*

We evaluated provenance on two programs: Query capturing and output Explaining.

Query Lineage Capture A data provenance system should be able to track down the input records used in a query. To evaluate KAKUTE in capturing query lineage, we wrote a program `MovieQuery`, which queries on a movie database (IMDB) and analyses movie data. When we got statistics of movies, we also got the lineage of each result record and people who watched the movie.

Output Explain To explain the output of a algorithm for some specific input, it is essential to show which input is used to produce a output. We used the ConnectedComponent (§8) as example for explaining output [8]. To explain the output, it is necessary to tell why two nodes are connected. we added tag to edges, and this tag only propagated to other nodes when their labels had been propagated to other nodes. Thus, the back-tracing edges for an output label ensembled a path to the original node with the label. On the other hand, forward tracing an input record were used for evaluating how important an edge was for composing a connected component in a graph.

8.5.3 *Program Bugs.*

We evaluated our system's applicability in debugging by 4 cases.

Illegal Input In a big-data program, some data may be in an illegal format, and the program may fail to process these data. We generated some illegal inputs (in an illegal format) for our log analysis program, according to an example in a previous work [17]. This program showed parsing error as this Illegal Input was not what the system intended. KAKUTE identified problematic inputs by back-tracing problematic records.

Task Failure Big-data programs may contain bugs in a UDF, and these bugs may only show for some inputs. We added some code in a `WikipediaPageRank` program like a previous paper [11], which parses a Wikipedia dataset, and gets edges and nodes from the dataset. With these problematic code, some tasks stopped unintentionally with XMLParsingError. Our debugger got the records and the UDFs, then re-ran the UDFs with these records. As a result, we found out that some codes used `\n` to represent an empty article.

Coverage Checking Big-data programs may fail to make use of all or most input to produce the final output. To identify this problem, we need to check the percentage of input used to produce all output (i.e., input coverage). We used a `KMeans` algorithm that clusters flower regarding their features. After coverage tests, we found out that only half of the data was used in computation, showing potential programming bugs. Therefore, we ran the program with only the non-covered result, and we found out that a record was ignored when their axises of the first dimension were 0. We ran the program after fixing the bug, and we were able to get all data covered.

Flow Verification A programmer can verify if a program is written correctly by verifying the relation between output and input (e.g., a output filed needs a specific input field). We wrote a MatrixMutiplication that accidentally multiples a row with another row. We defined our own rule for verifications and the verifier showed the program was wrong. After that, we modified the program and got the right program.

8.5.4 ***Performance Bugs.*** A program with naive partition scheme can result in much more shuffle traffics than a well-designed one [45], causing a much higher computation time. To generate a better partition scheme, we ran 4 programs with a small subset of data. The 4 programs were Matrix Multiplication, AdjcantList, SparkPageRank and ConnectedComponent. Among all applications, shuffle size have been reduced on average by 11%. We checked the dependency of each UDF, and we found out that AdjcentList had dynamic functional dependencies, while other three kept a static functional dependency. SparkPageRank and ConnectedComponent partition data on the whole edge tuple (in, out) instead of only "in", causing more shuffle traffic in `distinct`.

8.6 Limitation

Like previous information flow tracking systems [3, 14], Kakute only tracks data flows and ignores control flows with the concerns of the high overhead of control flow tracking incurred in previous systems [6, 9]. Previous work [1] makes use of static analysis [32] to prevent attacks through control flow. This technique can be a compliment to our dynamic flow tracking system.

There are other side channel attacks such as timing channel attacks that Kakute can not handle currently. These attacks are out of our design goal and previous work [4] has addressed these problems. Kakute does not support tracking of broadcast variables in Spark, as it exceeds the scope of UDFs, we plan to intercept the broadcast procedure and track broadcast variables in future implementations.

9 RELATED WORK

DISC Provenance Provenance has shown wide range of applications including debugging [17] and network management [48]. RAMP [18], Newt [25], Pig [16] and Titian [19] adopt a record-level tracking approach for data provenance in DISC frameworks. Chothia [8] introduces a novel framework for output explaining in iterative programs with differential dataflow abstraction. These systems also adopt the record-level tracking approach, so they have low precision in programs containing many-to-many transformations. Kakute, however, provides fine-grained IFT in terms of field level, and is precise and general for data provenance.

Information Flow Tracking IFT has been proposed to tackle security problems [33, 41], debugging [24], and program analysis. Cloudfence [35], Pileus [39] and Taint-Exchange [44] propose doing IFT across processes and hosts to secure cloud services. SilverLining [22] introduces IFT in a job level in Mapreduce. Neon [47] tries to migrate IFT to virtual machines. However, no IFT system exists for DISC frameworks. Kakute applies IFT to DISC frameworks, providing useful primitives for improving reliability of programs.

DISC Program Debugging Data-intensive Computation program is difficult to debug [17, 25], Arthur [11] introduces a framework with features like breakpoint, backward and forward tracking for Spark. BigDebug [17] leverages Titian to provide debugging primitives for Spark. Kakute takes field-level tracking, which provides fine-grained information for debugging. Kakute can be a complement to these system, yet it can also be applied to auto-testing of the distributed programs.

DISC Security DISC security is an emerging issue as more and more user sensitive data is captured and processed by distributed systems. Airavat[37], PINQ [28] and GUPT [30] propose to apply differential privacy [13, 29] in Mapreduce, to prevent leakage from user query, but differential privacy can result in incorrect results. Sedic [46] proposes to offload sensitive computations to private clouds. MrLazy [1] proposes a framework of combining data provenance and Static IFT of UDF, to provide fine-grained information flow for security. However, static IFT is not precise and may suffer from false positive. Kakute provides fine-grained information control of sensitive data, with no need to modify the original program.

10 CONCLUSION

We have presented Kakute, the first precise and fine-grained IFT system for security and reliability problems in DISC frameworks. Kakute provides field-level dataflow with unified APIs, which is useful for various applications. We have proposed two techniques, Reference Propagation and Tag Sharing, to get efficient IFT in big data. Kakute is fast and precise with different granularities of IFT, and evaluation has been done on a wide range of algorithms, datasets and bugs. Kakute can greatly improve the security and reliability of big-data.

ACKNOWLEDGMENTS

The authors thank all anonymous reviewers for their valuable comments and feedbacks. The work is funded in part by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (HK RGC ECS No.27200916, HK RGC GRF No.17207117) and a Croucher innovation award.

REFERENCES

[1] S. Akoush, L. Carata, R. Sohan, and A. Hopper. Mrlazy: Lazy runtime label propagation for mapreduce. In *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'14, pages 17–17, Berkeley, CA, USA, 2014. USENIX Association.

[2] M. R. Asghar, M. Ion, G. Russello, and B. Crispo. Securing data provenance in the cloud. In *Open problems in network security*, pages 145–160. Springer, 2012.

[3] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM.

[4] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[6] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 463–475. IEEE, 2007.

[7] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 401–412. IEEE, 2008.

[8] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *Proceedings of the VLDB Endowment*, 9(12):1137–1148, 2016.

[9] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[10] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The International Journal on Very Large Data Bases*, 12(1):41–58, 2003.

[11] A. Dave and M. Zaharia. Arthur: Rich post-facto debugging for production analytics applications.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[13] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, pages 265–284, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–6, 2010.

[15] M. Ganai, D. Lee, and A. Gupta. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 46. ACM, 2012.

[16] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, Aug. 2009.

[17] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 784–795, New York, NY, USA, 2016. ACM.

[18] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR 2011*. Stanford InfoLab.

[19] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. *Proc. VLDB Endow.*, 9(3):216–227, Nov. 2015.

[20] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2013.

[21] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[22] S. M. Khan, K. W. Hamlen, and M. Kantarcioglu. Silver lining: Enforcing secure information flow at the cloud edge. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 37–46. IEEE, 2014.

[23] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[24] T. R. Leek, G. Z. Baker, R. E. Brown, M. A. Zhivich, and R. Lippmann. Coverage maximization using dynamic taint tracing. Technical report, DTIC Document, 2007.

[25] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 17. ACM, 2013.

[26] A. P. Martin, J. Lyle, and C. Namiluko. Provenance as a security control. In *TaPP*, 2012.

[27] P. McDaniel. Data provenance and security. *IEEE Security & Privacy*, 9(2):83–85, 2011.

[28] F. McSherry. Privacy integrated queries. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, Inc., June 2009.

[29] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 94–103, Washington, DC, USA, 2007. IEEE Computer Society.

[30] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. Gupt: Privacy preserving data analysis made easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 349–360, New York, NY, USA, 2012. ACM.

[31] P. K. Murthy. Top ten challenges in big data security and privacy. In *Test Conference (ITC), 2014 IEEE International*, pages 1–1. IEEE, 2014.

[32] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[33] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[34] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[35] V. Pappas, V. P. Kemerlis, A. Zavou, M. Polychronakis, and A. D. Keromytis. Cloudfence: Data flow tracking as a cloud service. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 8145*, RAID 2013, pages 411–431, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[36] https://cwiki.apache.org/confluence/display/PIG/PigMix.

[37] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[38] https://spark.apache.org/examples.html.

[39] Y. Sun, G. Petracca, X. Ge, and T. Jaeger. Pileus: Protecting user resources from vulnerable cloud services. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 52–64, New York, NY, USA, 2016. ACM.

[40] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: limiting mobile data exposure with idle eviction. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 77–91, 2012.

[41] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.

[42] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[44] A. Zavou, G. Portokalidis, and A. D. Keromytis. Taint-exchange: A generic system for cross-process and cross-host taint tracking. In *Proceedings of the 6th International Conference on Advances in Information and Computer Security*, IWSEC'11, pages 113–128, Berlin, Heidelberg, 2011. Springer-Verlag.

[45] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions.

[46] K. Zhang, X. Zhou, Y. Chen, X. Wang, and Y. Ruan. Sedic: privacy-aware data intensive computing on hybrid clouds. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 515–526. ACM, 2011.

[47] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System support for derived data management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 63–74, New York, NY, USA, 2010. ACM.

[48] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proceedings of the VLDB Endowment*, volume 6, pages 49–60. VLDB Endowment, 2012.