1 Research Background

This section introduces the background of relevant techniquest in this proposal.

1.1 Big-data Computing Frameworks

DISC frameworks (e.g.Spark [58], DryadLINQ [57], MapReduce [14]) 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. large-scale Shuffles are frequent and sometimes are performance bottlenecks in DISC.

To avoid excessive computation, many DISC systems adopt the lazy transformation approach. [35, 57, 58]. Spark uses lazy transformations (e.g.MAP) for efficiency, and calls to these transformations 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 (§2.1.2) leverages the lazy transformation feature in DISC to present its new Reference Propagation technique.

1.2 DFT and Diff Privacy

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 [27].

Multiple research has been focusing on efficiency and applications of IFT. Shadowreplica [24] proposed to make use of the multicore resources while SHIFT [7] suggests accelerating dataflow tracking with hardware support. Several research [18, 29] 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 [16, 50], providing secure cloud services [38] and server program runtime [27]. To the best of our knowledge, no IFT system exists for big-data.

1.3 Intel SGX

Trusted Execution Environment (TEE) is a promising technique that protects computation on cloud even through the operating system is compromised. The program is running in a secure environment, and memory can not be seen by malicious parties. For example, Intel-SGX [22] runs programs in a enclave, which is protected and can not be see by the system.

However, running programs in a enclave needs modifications to the original programs, which needs a great effort and is error-prone. Recent work [5, 59] run Zookeeper and SparkSQL in enclaves, and both of them rewrote codes running in enclaves using C++. BigMatrix [43] proposes a secure and oblivious vectorization abstraction for Python, but it also needs modifications to the original programs. These methods causes two problems. First, modifications are necessary to run programs in enclaves. Second, running C++ in JVM breaks the protections provided Java.

1.4 Related work by others

1.5 Software-based Privacy-preserving Analytic

Homomorphic encryption [15, 20, 36] is a software-based techniques for protecting data in untrusted environments. Homomorphic encryption can be sorted as two kinds: Fully homomorphic encryption (FHE) and partial homomorphic encryption. Partial homomorphic encryption (e.g. Additive Homomorphic Encryption [36]) incurs a much lower overhead compared with FHE. A evaluation [19] on FHE shows a $10e9$ slowdown, which is acceptable in practice. Systems that adopts PHE (e.g. Monomi [53], Crypsis [49], CryptDB [40], MrCrypt [51]) reports a much better overhead, but it has limited expressiveness (e.g. SQL operators) and requires extra trusted servers for computations. Seabed [37] proposes asymmetric encryption schemes and reduces performance overhead incur by AHE, but it still has limited expressiveness.

1.6 Hardware-based Privacy-preserving Analytic

Intel SGX is a promising technique to provide privacy-preserving analytic in public clouds. Compared with software-based solutions, hardware-based solutions incurs much lower overhead. TrustedDB [3] is a hardware-based secure database. VC3 [42] proposes a secure distributed analytic platform with read-write validations on Mapreduce [14]. Opaque [59] supports secure and oblivious SQL operators on SparkSQL [2]. However, all these systems have limited expressiveness (e.g.SQL operators), and VC3 even needs to rewrite the program with C++. A recent work [34] proposes a oblivious machine leaning framework on trusted processors. BigMatrix [43] proposes an oblivious and secure vectorization abstraction on python, but it has limited expressiveness and it needs to rewrite the original program with this new abstraction. Although BigMatrix provides guideline for writing a oblivious program, but it would be a time-consuming and error-prone process.

1.7 Related work by the PI and co-I

The PI is an expert on secure and reliable distributed systems [9–12, 17, 25, 55, 56]. The PI's works are published in top conferences on systems (OSDI, SOSP, SOCC, TPDS, and ACSAC) and programming languages (PLDI and ASPLOS). The co-I is an expert on high-performance computing [1, 6, 26, 32, 60], fault-tolerance [44, 45], and VMs [28, 46, 54]. The co-I's works are published in top systems conferences (Cluster '02, SC '13, and ICPADS '14) and journals (JPDC '00, TPDS '13, IEEE Tran. Computers '14). As preliminary works for this proposal, the PI and co-I have developed Kakute [25] and TPDS [17] (parts of **Objective 1**).

2 Research Plan and Methodology

This section first proposes the three objectives in this proposal.

2.1 Objective 1: preventing big-data leakage with Precise IFT

This section presents PAXOS performance problem (§2.1.1) and KAKUTE (§2.1.2).

2.1.1 Problem: existing IFT systems are slow and incomplete for big-data

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 [4] 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 (§1.1). 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 [38] 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.

2.1.2 KAKUTE: a fast, precise IFT system for big-data

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

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

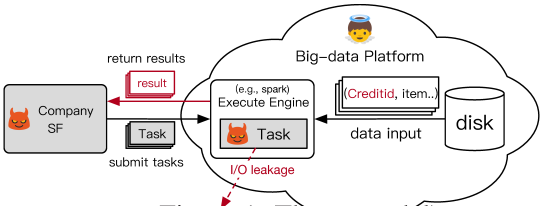

Figure 1: Threat model).

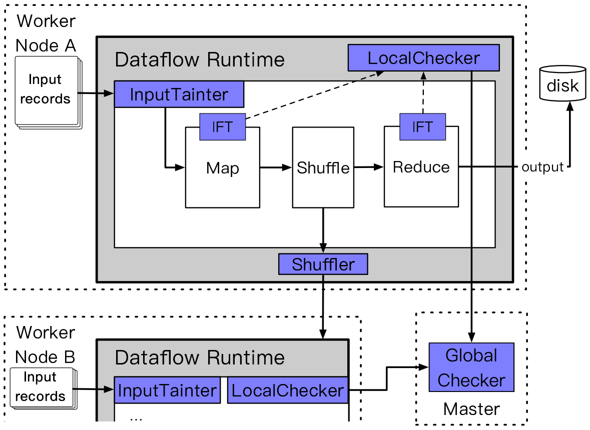

Figure 2: KAKUTE architecture.

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 will implement KAKUTE in Spark. We leverage Phosphor [4], 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.

Preliminary results. We evaluated KAKUTE on seven diverse algorithms, including three text processing algorithms WordCount [48], WordGrep [30] and TwitterHot [48], two graph algorithms TentativeClosure [48] and ConnectComponent [48], and two medical analysis programs MedicalSort [39] and MedicalGroup [39]. These algorithms cover all big-data algorithms evaluated in two related papers [21, 23]. We evaluated these algorithms with real-world datasets that are comparable with related systems [8, 21, 23]. We compared KAKUTE with Titian [23], a popular provenance system, on precision and performance. Our evaluation shows that: (1) KAKUTE is fast. Kakute had merely 32.3% overhead with INTEGER tag, suitable for production runs; and (2) KAKUTE is precise; it effectively detected 13 real-world security and reliability bugs presented in other papers [13, 21, 41].

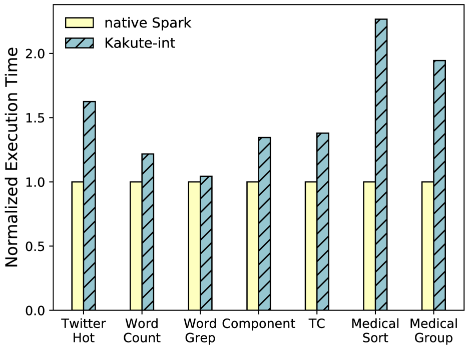

Figure 3: KAKUTE execution time normalized to Spark executions.

Future work. We will further improve the practicality of KAKUTE in two dimensions. First, we will extensively evaluate its performance on broad big-data programs. Second, we will leverage it to improve existing security techniques, including anonymization techniques and diff privacy (Objective 2).

Algorithm 1: Naive Aggregation Model

```
Input: Dataset T, dataset size N, privacy budget $\varepsilon_k$ for security level k, output range (min, max)
n = 4
for $i \leftarrow 1$ to $n$ do
    $O_i \leftarrow f(T_i)$;
    if $O_i$ > max, $O_i \leftarrow$ max if $O_i$ < max, $O_i \leftarrow$ min
for dimension $j$ of the output $O$ do
    k $\leftarrow max(getLevel(O_j))$;
    $O_j \leftarrow \frac{1}{n}\sum_{i=1}^{n} O_{ij} + Lap(\frac{max-min}{\varepsilon_k})$
Output: $O$
```

2.2 Objective 2: fine-grained diff privacy

We are proposing to combine IFT and differential privacy. With IFT, security levels are propagating throughout computation, while differential privacy is for preventing individual information leakage. In this model, different records and fields can have various security levels, therefore, a general and fine-grained differential private framework can be achieved. Programmer does not need to modify their analysis programs and tradeoff between usability and security can be achieved with different users.

It needs to determine the relation of the privacy budget and the security level. In a simplest model, there are 5 security levels: insensitive, dp-1, dp-2, dp-3, non-released. Insensitive record can be release directly, while non-released can not be used in any computation to the final result. dp-1 to dp-3 varies in terms of their privacy budgets for differential privacy. We may also consider the (δ, ε)-differential privacy model.

To estimate the range of a output, so that the laps noise distribution can be determine by running the same, differentially private percentile estimation algorithm on the inputs to compute the 25-th and the 75-th percentile privately (a.k.a, lower and upper quartiles) for the inputs.

We extend the differential privacy model developed in previous work [47]. We introduce two noise aggregation model that make use the fact that different record may have different security level (protection level).

A simplest model is to use the ε inferred by the highest security level, then add noise to the output directly. Add noise to the output with $f(x) + Lap(\frac{max-min}{\varepsilon_{max}})$. This simple model may generate two much noise to the final result, which make the result unusable. The complete algorithm is showed in Algorithm 1.

In this model, data is partitioned into multiple parts (size of each part is m). Each partition may consists different level. The noise aggregator adds noise to final result of each partition according to the security level of each partition.

Formally, data is partitioned into multiple parts denoted as p_1, p_2, ..., p_n. The security level and its corresponding ε are ε_1, ε_2, ..., ε_n. Therefore, the final aggregated result is

$$\frac{\sum_{i=1}^{n} f(x_i) + Lap(\frac{max_i - min_i}{\varepsilon_i})}{n} \tag{1}$$

The complete algorithm is showed in Algorithm 2.

The error of the final result incurs in this aggregator comes from two parts: the Laps noise error and the partition error. It is crucial to reduce the final error incurs by this aggregator while keeping the differential security guarantee. The error of this model (when the eps is the same) is

Algorithm 2: Combined Aggregation Model

```
Input: Dataset T, dataset size N, privacy budget $\varepsilon_k$ for security level k, output range (min, max)
n = 4
for $i \leftarrow 1$ to $n$ do
    $O_i \leftarrow f(T_i)$;
    if $O_i$ > max, $O_i \leftarrow$ max if $O_i$ < max, $O_i \leftarrow$ min for dimension j of the output O do
        k $\leftarrow max(getLevel(O_{ij}))$;
        $O_{ij} \leftarrow O_{ij} + Lap(\frac{max - min}{\varepsilon_k})$
foreach partition i of the output O do $O_j \leftarrow \frac{1}{n}\sum_{i=1}^{n} O_{ij}$;
Output: $O$
```

$$|\frac{1}{n}\sum_{i=1}^{n} f(T_i) - f(T)| + \frac{1}{n}\sqrt{2}\triangle f \sum_{i=1}^{n} \frac{1}{\varepsilon_i} \tag{2}$$

, and it should be minimized.

As final result of each partition may consist different security levels, the final noise can be less than previous method.

If we can get the statistic of security levels, we can estimate the block size in a better way. In specific, suppose there are k security levels and M_k means that M_k partitions has a maximum security level as k. The the above equation can be rewritten as

$$|\frac{1}{n}\sum_{i=1}^{n} f(T_i) - f(T)| + \frac{1}{n}\sqrt{2}\triangle f \sum_{i=1}^{k} \frac{1}{\varepsilon_i} * M_i \tag{3}$$

M_k can be calculated according to the total partition number n and the number of block that has security level as k (denoted as p_k), then we can further formulate it as:

2.3 Objective 3: building a secure just-in-time compiler

In the project, we plan to tackle the problem above. We are proposing to run unmodified Java program in enclaves to protect computation in public clouds or untrusted servers. We will design a Just-In-Time compiler for JVM and run secured functions in enclaves.

Thread Model We evaluate the program setup in public clouds. In a public cloud, only data provider, and a portion of code in the analytic platform that is running in the enclaved are trusted. In specific, the JIT compiler and the secure functions should be trusted. All other components, including operating system, hypervisor are trusted. Figure 4 shows the model (TODO).

Figure 5 shows the architecture of the system. Programmers annotate some functions as secure, so code of the functions and data processed by these functions should be kept as secrets. The annotated secure functions are compiled and executed inside enclaves so that data and code will not be leaked. The Java byte-codes of these function are compiled to native enclave codes, and are executed inside enclaves upon function calls.

There are two challenges that we need to address in our design: running unmodified Java programs with minimum TCB and reducing excessive enclave transitions.

A straightforward approach to run unmodified Java programs in enclaves for code and data protections is to run the whole JVM in enclaves. However, as argued in a previous work [5], it will blow up the TCB and cause a high overhead by running the whole JVM in enclaves.

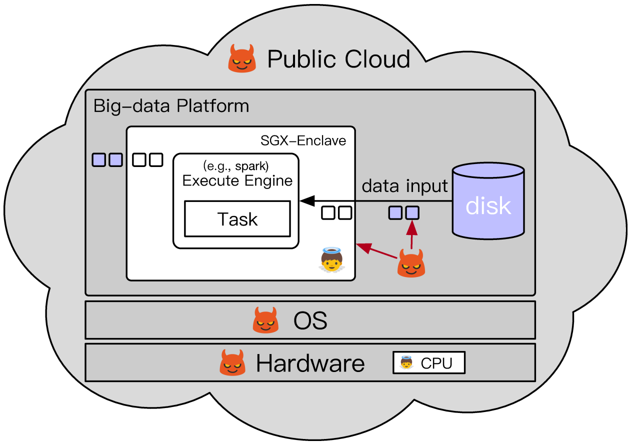

Figure 4: Threat model in public cloud. All grey boxes are not trusted and red lines represent potential leakages or attacks. Data is encrypted outside enclave and is in blue.

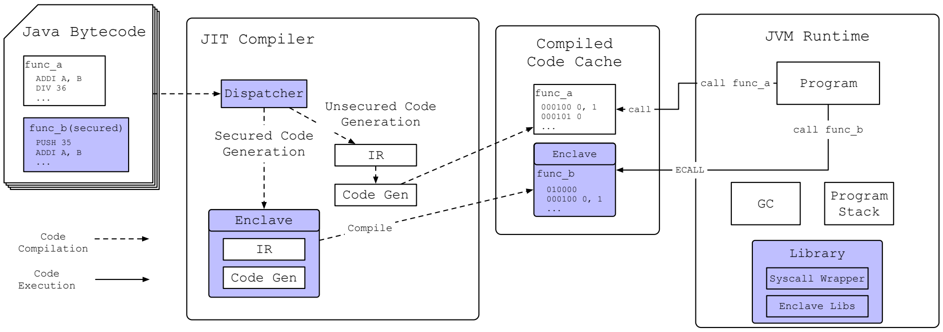

Figure 5: Architecture of Running Java with SGX. Some of the functions are annotated as secured function and they are compiled by a JIT in a enclave. The compiler compiles the functions into enclave code, so that the program will call into a enclave when calling into this function.

SGX is for protecting data and code running in enclaves, but code inside enclaves also have accesses to the regular memory region. Therefore, code in the enclaves can write sensitive data to unprotected memory regions when the code is compromised. The OpenJDK implementation of JVM contains up to millions lines of code, which is unpractical for formal verifications as it is time-consuming with current verification methods (§1.4).

Intel SGX contains a protected memory region call Enclave Page Cache (EPC), and evictions from this cache cause expensive encryption costs. Although next generation of SGX [31] may support a larger EPC, current generation of SGX only has an EPC up to 128MB. In practice, only around 90 MB can be allocated. Therefore, running programs with large memory consumption will cause much higher overhead. A recent work [59] shows that running program below this limit incur an only 7.46% overhead, while slightly exceeding this limit causes 50% to 60% overhead. Therefore, we have to reduce the TCB size for running unmodified Java programs in enclaves.

To reduce TCB size, we propose to run only some programmer-annotated functions in enclave. To this end, we propose a split execution framework. Only the Intermediate Representation (IR) and Code Generation model are trusted in the system. There will be two instances of compilers for compiling Java bytecode, one for secure code compilations and one for untrusted code compilations. Therefore, the TCB is greatly reduced, as we do not need to trust the JVM runtime (e.g. Garbage Collection). Also, as only some functions should be executed securely, the split execution framework should not cause high performance overhead.

When calling into and out of a function in enclaves, an ECALL and OCALL will be invoked in the CPU. In specific, the CPU is trapped into a new mode (except from the User and Privileged mode), and the current frame is encrypted and saved. A recent work [52] shows that enclave transitions are 60X slower than system calls and several hundreds times slower than user function calls. Moreover, encryption and decryption are required for secure function calls, which makes enclave transitions even more time-consuming. Our evaluation on a recent privacy-preserving data analytic system [59] shows that it incurs 3.4k transitions for processing 10k data (for two operations select and groupBy). In fact, these processing functions can be pipelined and processed in a single enclave function. Our project takes reducing transitions as one of our targets.

We introduce two techniques, Cost-based Compilation and Asynchronous Enclave Call, to tackle this challenge. Cost-based Compilation can make use of the JVM hotspot features, and analyse the hotspot enclave functions. It builds a tree of callers and callee of functions, and combines two enclaves if the marginal benefit of combining them is larger than the transition cost. Initially, only function b and d are running in enclaves, but the compiler finds out that cost of running c in an enclave is less than the transition cost (assume it to be 2), then the whole function a will be compile as an enclave. In another case where running function c takes a high cost, combinations of enclaves will not happen. Therefore, transition of enclaves can be reduced. This technique can be applied online or offline. In a offline version, it sample the program and finish the optimisation offline.

Asynchronous Enclave Call convert the synchronous enclave calls to asynchronous enclave calls. In specific, when a secure function is called, the function call and its parameters are put into a QUEUE which will be fetch by the enclave can be executed. After finishing execution, the result will be stored in QUEUE_R and the current execution will resume and continue. This technique makes use of the multi-core hardware architecture and enclaves and the main program are running in separate threads.

These two technique can be applied simultaneously, the compiler can run the sample and optimisation offline. After that, the compiler run a enclave and the program in separate threads to avoid transition of enclaves.

2.4 Research timeline

This project will require two PhD students S1 and S2 to work for three years. In the first year, S1 will design and fully implement the KAKUTE system (part of **Objective 1**), and S2 will evaluate its performance and robustness on various real-world big-data queries (part of **Objective 1**). In the second year, S1 will use KAKUTE to fully develop the proposed fine-grained privacy model (part of **Objective 2**), and S2 will implement the two algorithms proposed for this model (part of **Objective 2**). In the third year, S1 will build the secure big-data compiler **Objective 3** and S2 will evaluate this compiler on diverse real-world big-data queries.