1 Research Background

This section presents the background of big-data frameworks (§1.1), sofware-based privacy techniques (§1.2), hardware-based privacy techniques (§1.3), motivation (§1.4), and related work (§1.5 and §1.6).

1.1 Big-data Computing Frameworks

Big-data frameworks (e.g., Spark [63] and MapReduce [14]) are popular for computations on tremendous amounts of data. These frameworks provide self-defined Java functions (e.g., MAP/REDUCE) to let computational providers write their algorithms, and they and automatically apply these functions on the data stored across computers in parallel.

To avoid excessive computation, big-data frameworks adopt a lazy transformation approach [38, 62, 63]. Spark often uses lazy transformations (e.g., MAP), and calls to these transformations only create a new data structure called RDD with *lineage* (the sequence of transformations for a data record). The actual transformations are only triggered when collecting operations (e.g., COLLECT, COUNT) are called. These collecting operations trigger transformations along lineages, so unnecessary computations are avoided. **Objective 1** will leverage lazy transformation to create a fast DFT technique called Reference Propagation (§2.1).

1.2 Software-based Privacy Techniques

Data Flow Tracking (DFT) is a mandatory access control technique for preventing sensitive information leakage [36]. DFT attaches a tag to a variable (or object), and this tag will propagate throughou computations on the variable at runtime. DFT has been applied to various areas, such as preventing sensitive information (e.g.GPS data and contacts) leakage in cellphone [17, 54], web services [41], and server programs [27]. To the best of our knowledge, no DFT system exists for big-data computing.

Complimentary to DFT, statistical techniques, including K-anonymization methods [30, 53] and differential privacy [32, 35, 44], allow the aggregration of sensitive data while adding random noise to preserve individual privacy. However, these statistical techniques are either not secure (K-anonymization) or suffering from great losses of accuracy (differential privacy). A recent work [22] reports more than 30% losses of accuracy. For a query results, low accuracy means bad utility: a simple KMeans program will return centroids far from the accurate ones, and the accuracy loss rate is much larger than the training error rate which is several percents in practice.

A key reason for this bad utility problem is that differential privacy can not track how sensitive data fields flow to query results, so they have to take a coarse-grained approach, which conservatively adds noise to all fields and records. **Objective 2** (§2.2) proposes a novel fine-grained differential privacy technique, which combines the strengths of DFT and differential privacy.

1.3 Hardware-based Privacy Techniques

Trusted Execution Environment (TEE) is a promising technique for protecting computation in a public cloud even if the cloud's operating systems or hypervisors are compromised. For instance, Intel-SGX [23], a popular commercial TEE product, runs a program in a hardware-protected *enclave*, so code and data are protected from outside. Compared with the approach of computating on encrypted data (§1.5), TEE is much safer and 100X to 1000X faster. For instance, a SGX-based system Opaque [65] incurs a moderate performance overhead of 30% compared to native big-data queries.

However, to practically run Java big-data queries with SGX, two open challenges remain. First, existing SGX-based systems [65]requres computational providers to manually rewrite the readily pervasive Java queries into SGX-compatible C++, a time-consuming and error-prone process. Second, existing SGX-based systems for big-data have too large Trusted Computing Base (TCB). Existing systems (e.g., SGX-BigMatrix [46]) run a whole language interpreter (e.g., JVM and Python runtime) in enclaves, causing a too large (and too dangerous) TCB: JVM code comes from many different parties/vendors and extremely hard to verified. **Objective 3** (§2.3) takles these two open challenges by building a new just-in-time compiler.

1.4 Motivation of objectives

Data leakage is a top threat in cloud computing [**?**]. In a data provider's perspective, there have been severe real-world data privacy breaches [**?**] caused by computational providers and cloud providers. This proposal aims to preserve the data provider's privacy by going two directions. First, we propose two novel complimentary techniques in **Objective 1** (KAKUTE) and **Objective 2** (fine-grained differential privacy) to protect privacy against the computational providers in private clouds. Second, we propose **Objective 3** (a new privacy-preserving compiler) to protect privacy against the (public) cloud providers. By integrating the outcomes from all three objectives, data privacy will be effectively preserved.

1.5 Related work by others

Computing on encrypted data. Homomorphic encryption [16, 20, 39] is a technique for performing computations on encryted data in untrusted environments. Homomorphic encryption contain two kinds: Fully homomorphic encryption (FHE) and partial homomorphic encryption. Partial homomorphic encryption (e.g. Additive Homomorphic Encryption [39]) 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 [57], Crypsis [52], CryptDB [43], MrCrypt [55]) reports a much better overhead, but it has limited expressiveness (e.g., SQL operators) and requires extra trusted servers for computations. Seabed [40] proposes asymmetric encryption schemes and reduces performance overhead incur by AHE, but it still has limited expressiveness.

SGX-based systems. 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 [4] is a hardware-based secure database. VC3 [45] proposes a secure distributed analytic platform with read-write validations on Mapreduce [14]. Opaque [65] supports secure and oblivious SQL operators on SparkSQL [3]. However, all these systems have limited expressiveness (e.g.SQL operators), and VC3 even needs to rewrite the program with C++. A recent work [37] proposes a oblivious machine leaning framework on trusted processors. BigMatrix [46] 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.

Big-data privacy systems. Big-data privacy has been a top emerging threat as more and more user sensitive data is captured and processed in clouds. Airavat[44], PINQ [32] and GUPT [35] propose to apply differential privacy [15, 33] in Mapreduce, to prevent leakage from user query, but differential privacy can result in incorrect results. Sedic [64] proposes to offload sensitive computations to private clouds. MrLazy [2] proposes a framework of combining data provenance and static DFT analysis for self-defined queries, to provide fine-grained information flow for security. However, static DFT 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.

1.6 Related work by the PI and co-I

The PI is an expert on secure and reliable distributed systems [9–12, 18, 25, 59–61]. The PI's works are published in top conferences on systems (OSDI, SOSP, SOCC, TPDS, and ACSAC) and programming languages (PLDI and ASPLOS). Recently, the PI has collaborated with Huawei to launch a technology transfer project based on his dependable distributed system [60]. The co-I is an expert on high-performance computing [1, 7, 26, 34, 66], fault-tolerance [47, 48], and Java compilers [28, 49, 58]. The co-I's works are published in top systems conferences (Cluster, SC, and ICPADS) and journals (JPDC, TPDS, IEEE Tran. Computers). As preliminary results for this proposal, the PI has presented KAKUTE [25] in ACSAC '17, and the PI and co-I have collaborated to present CONFLUENCE [18] in TPDS '17.

2 Research Plan and Methodology

2.1 Objective 1: preventing big-data comput

This section presents major challenges in existing DF

2.1.1 Challenges: existing DFT systems are too sl

Although DFT is a powerful access control technique
head, especially for data-intensive computations. For
in Spark with a WordCount algorithm on a small data
computation time compared with the native Spark ex
big-data frameworks usually contain *shuffle* operation
ers. However, most existing DFT systems ignore datal
cross-host dataflows, transferring all tags in shuffles c

2.1.2 KAKUTE: a fast, precise IFT system for big

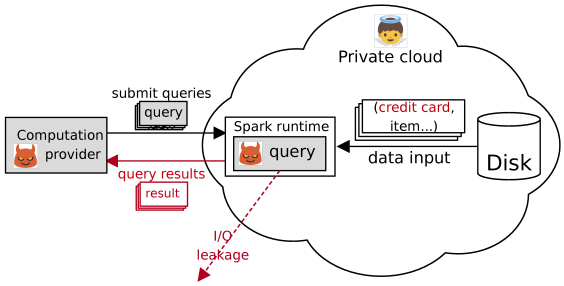

Figure 1: Threat model of KAKUTE. Red colors means sensitive data or leaking channels. Shaded (grey) components may leak data, and KAKUTE is designed to defend against them.

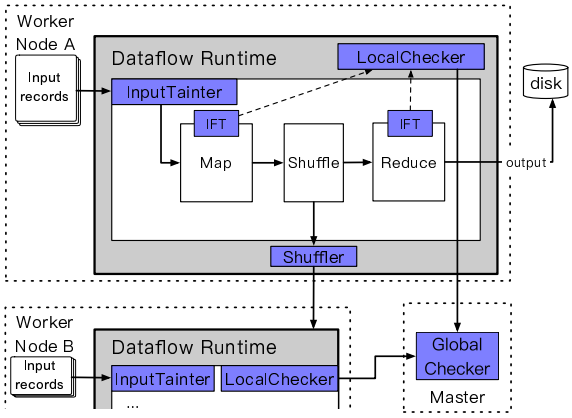

Figure 2: KAKUTE architecture.

We present KAKUTE, the first precise and complete DFT system for big-data frameworks. Our key insight to address the DFT performance challenge is that multiple fields of a record often have the same tags with the same sensitivity level. For example, in an Taobao order record ⟨t', ŭserId, p̆roductID⟩, only the ŭserId field is sensitive, while the other fields are insensitive and they can share the same tag. 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 self-defined queries, while Tag Sharing reduces memory usage by sharing tags among multiple fields in each record. To tackle the completeness challenge, KAKUTE completely captures dataflows in shuffles by intercepting the , and it efficiently reduces the amount of transferred IFT tags using Tag Sharing.

Figure 1 defines KAKUTE's threat model. Figure 2 shows KAKUTE's architecture design: the InputTainter component provides easy-to-use APIs for data providoers to automatically tag sensitive fields, the DFT component is enabled in self-defined functions, the Local- and Global-Checker components detect and prevent illegal flows of sensitive fields (e.g., credit cards flow to IO functions self-defined functions). Shuffle operations across computers are intercepted and added tags. Therefore, DFT is completely captured across computers.

We will implement KAKUTE and integrate it with Spark. We will leverage Phosphor [5], an efficient DFT system working in the Java byte-code level. KAKUTE instruments computations of a Spark worker process to capture dataflow inside self-defined-functions. KAKUTE provides different granularities of tracking with two types of tags: INTEGER and OBJECT tags. INTEGER provides 32 distinct tags for identifying 32 sensitivity levels, suitable for detecting data 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 will implement 4 built-in checkers for 4 security and reliability problems: sensitive information leakage, data provenance, programming and performance bugs.

Preliminary results. We have implemented a KAKUTE preliminary propotype and evaluated it on seven diverse algorithms, including three text processing algorithms WordCount [51], WordGrep [29] and TwitterHot [51], two graph algorithms TentativeClosure [51] and ConnectComponent [51], and two medical analysis programs MedicalSort [42] and MedicalGroup [42]. These algorithms cover all big-data algorithms evaluated in two related papers [21, 24]. We evaluated these algorithms with real-world datasets that are comparable with related systems [8, 21, 24]. We compared KAKUTE with Titian [24], a popular provenance system, on precision and performance. Our evaluation shows that: (1) KAKUTE is fast. Kakute had merely 32.3% overhead (Figure 3) with INTEGER tag, suitable for production runs; and (2) KAKUTE is precise; it detected 13 real-world security and reliability bugs presented in other papers [13, 21, 44].

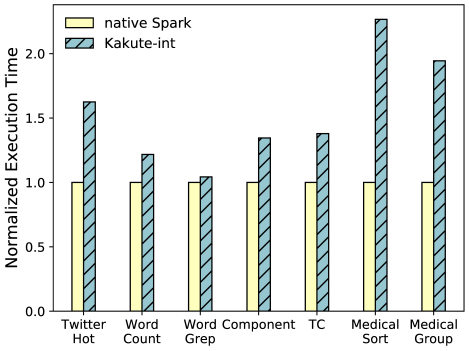

Figure 3: KAKUTE execution time normalized to native Spark executions. 100% means no overhead.

Future work. We will further improve and extend KAKUTE in two directions. First, we will extend KAKUTE to detect broader types of security bugs, including access patterns and timing channels. Second, will leverage KAKUTE to improve other complementary privacy techniques, including anonymization techniques and differential privacy (Objective 2).

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 [50]. 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

Algorithm 1: Naive Aggregation Model

Input: Dataset T, dataset size N, privacy budget ε_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

Algorithm 2: Combined Aggregation Model

Input: Dataset T, dataset size N, privacy budget ε_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

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

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

, 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).

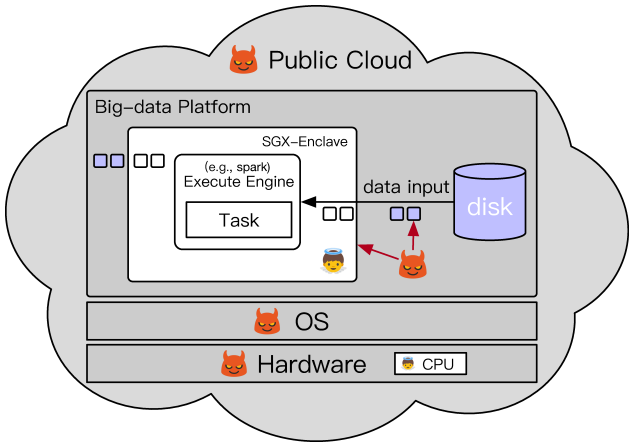

Figure 4: Threat model in public cloud. All grey boxes are not trusted and red lines represent potential leakages or attacks. Data is encrypted outside enclave and is in blue.

Figure 5 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 [6], it will blow up the TCB and cause a high overhead by running the whole JVM in enclaves.

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 (§1.5).

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

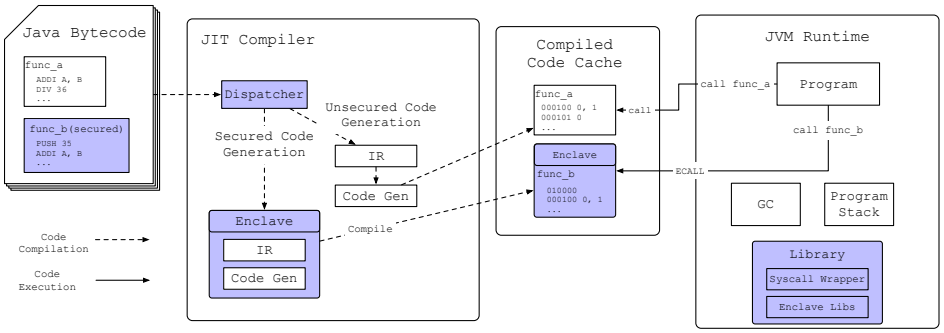

Figure 5: Architecture of Running Java with SGX. Some of the functions are annotated as secured function and they are compiled by a JIT in a enclave. The compiler compiles the functions into enclave code, so that the program will call into a enclave when calling into this function.

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 [65] 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 [56] 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 [65] 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.