Data Leakage Mitigation of User-Defined Functions on Secure Personal Data Management Systems

Robin Carpentier, Iulian Sandu Popa, Nicolas Anciaux

Data Leakage Mitigation of User-Defined Functions on Secure Personal Data Management Systems

Robin Carpentier
Univ. Versailles St-Q.-en-Yvelines
Versailles, France
robin.carpentier@uvsq.fr

Iulian Sandu Popa
Univ. Versailles St-Q.-en-Yvelines
Versailles, France
iulian.sandu-popa@uvsq.fr

Nicolas Anciaux
Inria Saclay Île-de-France
Palaiseau, France
nicolas.anciaux@inria.fr

ABSTRACT

Personal Data Management Systems (PDMSs) arrive at a rapid pace providing individuals with appropriate tools to collect, manage and share their personal data. At the same time, the emergence of Trusted Execution Environments (TEEs) opens new perspectives in solving the critical and conflicting challenge of securing users' data while enabling a rich ecosystem of data-driven applications. In this paper, we propose a PDMS architecture leveraging TEEs as a basis for security. Unlike existing solutions, our architecture allows for data processing extensiveness through the integration of any user-defined functions, albeit untrusted by the data owner. In this context, we focus on aggregate computations of large sets of database objects and provide a first study to mitigate the very large potential data leakage. We introduce the necessary security building blocks and show that an upper bound on data leakage can be guaranteed to the PDMS user. We then propose practical evaluation strategies ensuring that the potential data leakage remains minimal with a reasonable performance overhead. Finally, we validate our proposal with an Intel SGX-based PDMS implementation on real data sets.

1 INTRODUCTION

Successive steps have been taken in recent years to give individuals new ways to retrieve and use their personal data. In particular, the introduction of the right to data portability [20] allows individuals to retrieve their personal data from different sources (e.g., health, energy, GPS tracks, banks). Personal Data Management Systems (PDMSs) are emerging as a technical corollary, providing mechanisms for automatic data collection and the ability to use data and share computed information with applications. This is giving rise to new PDMS products such as Digi.me, Cozy Cloud, Personal Infomediaries [27], Solid/PODS [40] to name a few (see e.g., [7] for a recent survey), and to large initiatives such as Mydata.org supported by data protection agencies.

Context. PDMSs introduce a new paradigm for data-driven computations where specialized computation functions, written by third-parties, can be sent to the PDMS for execution. Only the result of the computation (but not the raw personal data used as input) is shared with the third-party. This paradigm is in contrast to traditional solutions, where the user's personal data is sent to a third-party application or service that performs the required computation. The example below illustrates this new paradigm.

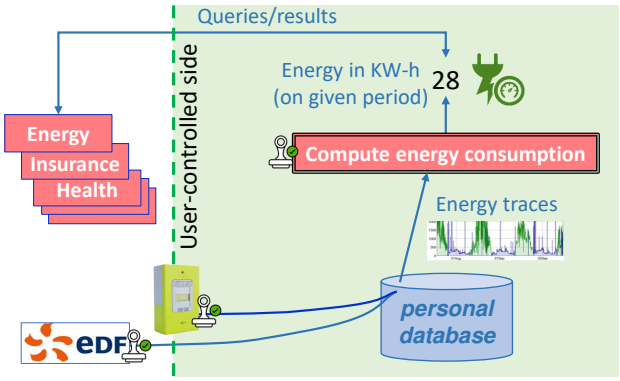


Figure 1: PDMS computation scenario ('Energy')

> ***'Energy' running example.*** An energy supplier wishes to offer services, precisely calibrated according to the energy consumption of its future customers. In order to establish a tailor-made offer, the provider needs to evaluate different statistical computations on the customer's consumption. Thanks to their PDMS offering confidentiality guarantees and a smart meter, customers agree to disclose these statistics but not their detailed consumption, and thus install the function sent by the supplier. The attestations provided by the PDMS even allow the supplier to commit to a quote since they obtain guarantees on the computation made.

This scenario is challenging since it calls for (i) *extensiveness* to ensure suppliers that ad hoc function code necessary to evaluate the desired results can be specified and used for their computation, and (ii) *security* to ensure customers that their detailed personal data is not disclosed to third parties (including their future supplier) outside of the sphere of control of their PDMS. Several similar scenarios are realistic, in other contexts than energy, to establish service offers based on statistical analysis of historical personal *factual* data related to a user, e.g., health services (based on medical and prescription history), car insurance (GPS traces), banking or financial services (bank records) or ecological services (ecological bonus based on mobility history).

Objective. Our goal in this paper is to solve this conflict between extensiveness and security for processing functions dealing with large volumes of personal data (typically, aggregation functions), whose code, defined by a third-party application querying the PDMS called *App*, is evaluated in the PDMS environment but cannot be considered fully trusted from the point of view of the *PDMS user* (owner of the data and of the PDMS). More precisely, we focus on controlling potential information leakage in legitimate query results during successive executions of such functions.

Limitations of existing approaches. Traditional DBMSs support user-defined functions (UDFs) to ensure extensiveness of computations. But their security relies on administrators, e.g., checking/auditing UDF code and their semantics. In contrast, a layman PDMS user cannot endorse this task nor rely on third-party administrators. The UDF code being actually implemented by an external App, it should be considered untrusted to the PDMS user. Furthermore, having authorized access to large volumes of personal data and the ability to externalize results to third-party App could raise privacy concerns and even be perceived as a risk of mass surveillance for PDMS users. An alternative approach is to employ information flow analysis techniques [33, 39, 41] to detect data leakage. But existing work essentially guarantees a non-interference property between the output of the computation and the sensitive inputs, which is not applicable to functions whose intrinsic goal is to produce aggregate results that depend on all sensitive inputs. Another approach would be to use anonymization (e.g., differential privacy [17, 38]) to protect the input of the computed function, but this method is not generic thus undermining the extensiveness property, and can hardly preserve result accuracy. Similarly, employing secure multiparty cryptographic computation techniques can hurt genericity (e.g., semi-homorphic encryption [19]) or performance (e.g., fully homomorphic encryption [21]), and cannot completely solve the problem of data leakage through legitimate results computed by untrusted code.

Proposed approach. We resort to a security model where trust relies on hardware properties provided by Trusted Execution Environments (TEEs), such as Intel SGX [14] or ARM TrustZone [35], and sandboxing techniques within enclaves, like Ryoan [28] or SGX-Jail [42]. Our approach consists in considering split execution techniques [11] between a set of 'data tasks' within sandboxed enclaves running untrusted UDF code on partitions of the input dataset to ensure extensiveness, and an isolated 'core' enclave running a trusted PDMS engine in charge of orchestrating the evaluation to mitigate personal data leakage and ensure security.

Contributions. We rely on [11] which formalizes the threat and computation models adopted in the PDMS context and introduces three security building blocks to bound the information leakage from user-defined computation results on large personal datasets. In this paper, we propose a set of execution strategies combining these building blocks to conciliate good performance and information leakage mitigation. Then, we validate our proposal using representative user-defined computations over two real-world datasets, on an SGX-based PDMS prototype platform.

The paper is organized as follows. Section 2 introduces the main components of the PDMS and the security properties considered. Section 3 introduces our computing and threat models and formulates the problem. Section 4 provides the security building blocks on which our proposal is based, analyzes their impact on information control and identifies an upper bound for information leakage at computation time. Section 5 gradually presents our execution strategies to mitigate information leakage while respecting good performance. Experiments are conducted in Section 6. Related work is presented in Section 7 and Section 8 concludes.

2 EXTENSIVE AND SECURE PDMS

We first present the PDMS architecture proposed in [7], to reconcile extensiveness and security. Next, we present the security properties we consider for the elements of this architecture preliminarily introduced in [10]. This background section is needed to formulate the problem (next section). We refer the reader to [7] for details about the logical architecture and to [12] for a concrete PDMS instance on Intel SGX.

2.1 Architecture Outline

Designing a PDMS architecture that offers security and extensiveness as defined above is a significant challenge due to a fundamental tension: security requires trusted code and environment to manipulate data, while extensiveness relies on user-defined, and thus potentially untrusted, code. We proposed in [7] a three-layers logical architecture to handle this tension, where *Applications* (Apps) on which no security assumptions are made, communicate with a *Secure Core* (Core) implementing basic operations on personal data, extended with *Data Tasks* (Data tasks) isolated from the Core and running user-defined code (see Figure 2), as described below:

Core. The Core is a secure subsystem that is a Trusted Computing Base (TCB). It is ideally minimal, inextensible –potentially proven correct by formal methods– and is isolated from the rest of the system. It constitutes the sole entry/exit point to manipulate PDMS data and retrieve results, and hence provides the basic DBMS operations required to ensure data confidentiality, integrity, and resiliency. It therefore implements a data storage module, a policy enforcement module to control access to PDMS data and a basic query module (as needed to evaluate simple selection predicates on database objects metadata to retrieve sets of authorized objects) and a communication manager for securing data exchanges with other layers of the architectures and with Apps accessing the PDMS.

Data tasks. Data tasks are introduced as a means to handle the code extensions needed to support user-defined functions kept isolated from the Core. Data tasks can execute arbitrary, application-specific data management code, thus enabling extensiveness (like UDFs in traditional DBMSs). The idea is to handle user-defined functions by (1) dissociating them from the Core into one or several Data tasks evaluated in a sufficiently isolated environment to maintain control on the data sent to them by the Core during computations, and (2) scheduling the execution of Data tasks by the Core such that security is globally enforced.

Apps. The complexity of these applications (large code base, extensible and not proven) and their execution environment (e.g., web browser) make them vulnerable. Therefore, no security assumption is made on applications, which manipulate only authorized data resulting from Data tasks but have no privilege on the raw data.

2.2 Security Properties

The specificity of our architecture is to remove from local or remote Apps any sensitive data-oriented computation, delegating it to Data tasks running UDFs under the control of the Core, within the PDMS user's environment. App leverages an *App manifest* which includes essential information about the UDFs to be executed by the App, including their purpose, authorized PDMS objects and size of results to be transmitted to the App. The manifest should be validated, e.g.

by a regulatory agency or the App marketplace, and approved by the PDMS user at install time before the App can call corresponding functions. Specifically, to maximize security our system implements the following architectural security properties:

P1. Enclaved Core/Data tasks. The Core and each Data task run in *individual enclaves* protecting the confidentiality of the data manipulated and the integrity of the code execution from the untrusted execution environment (application stack, operating system). Such properties are provided by TEEs, e.g., Intel SGX, which guarantees that (i) enclave code cannot be influenced by another process; (ii) enclave data is hidden from any other process (RAM encryption); (iii) enclave can prove that its results were produced by its genuine code [14]. Besides, the code of each Data task is sandboxed [28] within its enclave to preclude any voluntary data leakage outside the enclave by a malicious Data task function code. Such Data task *containment* can be obtained using existing software such as Ryoan [28] or SGXJail [42] which provide means to restrict enclave operations (bounded address space and filtered syscalls).

P2. Secure communications. All the communications between Core, Data tasks and Apps are classically secured using TLS (see Section 6) to ensure authenticity, integrity and confidentiality. Because the inter-enclave communication channels are secure and attested (e.g., establishing TLS channel with Intel SGX enclaves resorts to attestations), the Core can guarantee to Apps or third parties the integrity of the UDFs being called.

P3. End-to-end access control. The input/output of the Data tasks are regulated by the Core which enforces *access control* rules for each UDF required by an App as defined at the install time in the App manifest. Also, the Core can apply basic selection predicates to select the subset of database objects for a given UDF call. For instance, in our 'Energy' running example, a Data task running a UDF computing the consumed amount of energy during a certain time interval will only be supplied by the Core with the necessary energy consumption traces (i.e., the ones covering the given time interval). If several Data tasks are involved in the evaluation of a UDF, the Core guarantees the transmission of intermediate results between these Data tasks. Finally, the App only receives final computation results from the Core (e.g., the amount of energy consumed) without being able to access any other data.

Taken together, the above properties enforce the PDMS security and, in particular, the data confidentiality, precluding any data leakage, except through the legitimate results delivered to the Apps.

3 PROBLEM FORMULATION

To formulate the specific problem addressed in this paper, we introduce first our computation model leveraging UDFs and the considered threat model.

3.1 Computation Model

We seek to propose a computation model for UDFs (defined by any external App) that satisfies the canonical use of PDMS computations (including the use-cases discussed above). The model should not impact the application usages, while allowing to address the legitimate privacy concerns of the PDMS users. Hence, we exclude from the outset UDFs which are permitted by construction to extract large sets of raw database objects (like SQL select-project-join queries). Instead, we consider UDFs (denoted as a function f) with the following characteristics: (1) f has legitimate access to large sets of database objects and (2) f is authorized to produce various results of fixed and small size.

The above conditions are valid, for instance, for any aggregate UDF applied to sets of PDMS objects. As our running example illustrates, such functions are natural in PDMS context, e.g., to produce statistics using time series of user energy consumption within some given time intervals, or leveraging user GPS traces for statistics of physical activities, the traveled distance or the used modes of transportation over given time periods.

As illustrated by these examples, aggregates in a PDMS are generally applied on complex objects (e.g., time series, GPS traces, documents, images), which requires adapted and advanced UDFs at the object level. Specifically, to evaluate an aggregate function *agg*, a first function *cmp* needs to be computed for each object o of the input. For instance, *cmp* can compute the integral of a time series indicating the electricity consumption or the length of a GPS trace stored in o, while *agg* can be a typical aggregate function applied subsequently on the set of *cmp* results. Besides, we consider that the result of *cmp* over any object o has a fixed size in bits of $||cmp||$ with $||cmp|| << ||o||$ (e.g., in the examples above about time series and GPS traces, *cmp* returns a single value –of small size– computed from the data series –of much larger size– stored in o).

For simplicity and without lack of generality, we focus in the rest of the paper on a single App a and computation function f. Overall, our computation model is as follows:

Computation model. An App (or a third party) a is granted execution privilege on an aggregate UDF $f = agg \circ cmp$ with read access to (any subset of) a set O of database objects according to a predefined App manifest $\{< a, f, O >\}$ accepted by the PDMS user at App install time. a can freely invoke f on any $O_\sigma \subseteq O$, where σ is a selection predicate on some object metadata (e.g., a time interval) chosen at query time by a. The function f computes $agg(\{cmp(o)\}_{o \in O_\sigma})$, with *cmp* an arbitrarily complex preprocessing applied on each raw database object $o \in O_\sigma$ and *agg* an aggregate (or similar) function. We consider that both *agg* and *cmp* are deterministic functions and produce fixed-size results.

3.2 Threat Model

We consider that the attacker cannot influence the consent of the PDMS user, which is required to install UDFs. However, neither the UDF code nor the results produced can be guaranteed to meet the user's consented purpose. To cover the most problematic situations for the PDMS user, we consider an active attacker fully controlling the App a with execution granted on the UDF f. Thus, the attacker can authenticate to the Core on behalf of a, trigger successively the evaluation of f, set the predicate σ defining $O_\sigma \in O$ its input object set and access all the results produced by f.

Furthermore, since a also provides the PDMS user with the code of $f = agg \circ cmp$, we consider that the attacker can instrument the code of *agg* and *cmp* such that instead of the expected results, the execution of f produces some information coveted by the attacker, to reconstruct subsets of raw database objects used as input.

On the contrary, we assume that security properties P1 to P3 (see Section 2) are enforced. In particular, we assume that the PDMS

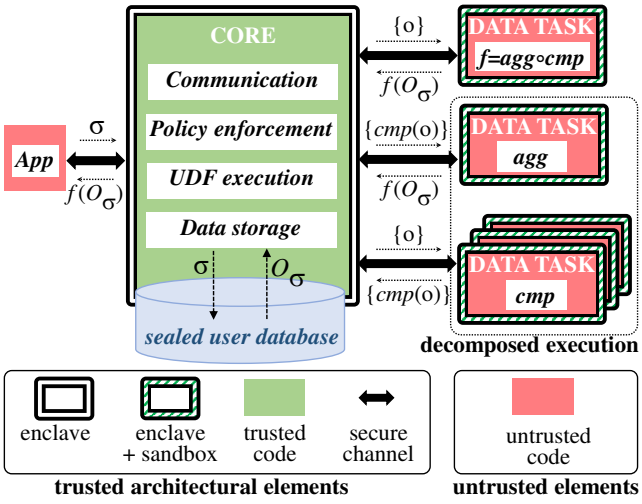


Figure 2: Computation and Threat Models.

Core code is fully trusted as well as the security provided by the TEE (e.g., Intel SGX) and the in-enclave sandboxing technique used to enforce P1 to P3. Figure 2 illustrates the computation model considered for the UDFs and the trust assumptions on each of the architectural elements of the PDMS involved in the evaluation.

Note that to foster usage, we impose no restrictions on the σ predicate and on the App query budget[1]. In addition, we do not consider any semantic analysis or auditing of the code of f since this is not realistic in our context (the layman PDMS user cannot handle such tasks) and also mostly complementary to our work as discussed in Section 7.

3.3 Problem Formulation

The precise goal of the paper is to address the following two questions: (1) Is there an upper bound on the potential leakage of personal information that can be guaranteed to the PDMS user, when evaluating a user-defined function f successively on large sensitive data sets, under the considered PDMS architecture, computation and threat models? (2) Is there a performance-acceptable execution strategy guaranteeing minimal leakage with potentially large volumes of personal data?

Answering these questions is critical to bolster the PDMS paradigm. A positive conclusion to the first question is necessary to justify a founding principle for the PDMS, insofar as bringing the computational function to the data (and not the other way around) would indeed provide a quantifiable privacy benefit to PDMS users. The second question may lead to a positive assessment of the realism and practical adoption of the proposed solutions.

Analysis of the problem requires appropriate quantification of leakage for attacks conducted using corrupted code, within the framework of the computational and threat models described earlier. Before presenting our simplified metric and problem formulation, let us consider a simple attack example:

[1]Such restrictions can be indeed envisioned for specific Apps and will be studied in our future work.

Attack example. The code for f, instead of the expected purpose which users consent to (e.g., analyzing energy consumption traces to compute required statistics), implements a function f_{leak} that produces a result called *leak* of size $||f||$ bits ($||f||$ is the number of bits allowed for legitimate results of f), as follows: (i) f sorts its input objects set O, (ii) it encodes on $||f||$ bits the information contained in O next to the previously leaked information and considers them as the –new– *leak*; (iii) it sends –new– *leak* as the result.

In a basic approach where the code of f is successively evaluated by a single Data task DT^f receiving a same set O of database objects as input, the attacker obtains after each execution a new chunk of information about O encoded on $||f||$ bits. The attacker could thus reconstruct the complete set O by assembling the received *leak*, after at most $n = \frac{||O||}{||f||}$ successive executions, with $||O||$ the size in bits needed to encode the information of O.

To address the first question, we introduce metrics inspired by traditional information flow methods (see e.g., [39]). We denote by $||x||$ the amount of information in x, measured by the number of bits needed to encode it. For simplicity, we consider this value as the size in bits of the result if x is a function and as its footprint in bits if x is a database object or set of objects. We define the leakage $L_f(O)$ resulting from successive executions of a function f on objects in O allowed to f, as follows:

Definition 1. Data set leakage. The successive executions of a function f, taking as input successive subsets O_σ of a set O of database objects, can leak up to the sum of the leaks generated by the non identical executions of f. Two executions are considered identical if they actually produced the same result on the same input (as the case for functions assumed deterministic). Successive executions producing n non-identical results generate up to a *Data set leakage* of size ${L_f}^n(O) = ||f|| \times n$ (i.e., each execution of f may provide up to $||f||$ new bits of information about O).

To quantify the number of executions of f required to leak given amounts of information, we introduce the *leakage rate* as the ratio of the leakage on a number n of executions, i.e., $\overline{{L_f}^n} = \frac{1}{n} \cdot {L_f}^n(O)$.

The above leakage metrics express the 'quantitative' aspect of the attack. However, attackers could also focus their attack on a (small) subset of objects in O that they consider more interesting and leak those objects first, and hence optimize the use of the possible amount of information leakage. To capture 'qualitative' aspect of an attack, we introduce a second leakage metric:

Definition 2. Object leakage. For a given –targeted– object o, the *Object leakage* denoted ${L_f}^n(o)$ is the total amount of bits of information about o that can be obtained after executing n times the function f on sets of database objects containing o.

The challenge is to propose execution strategies for evaluating untrusted user-defined functions in the PDMS context that on the one hand limit Data set and Object leakages (metrics above) to small values and on the other hand are efficient and implementable in practice. To address this problem, we proceed in two steps: (1) we introduce in Section 4 countermeasure building blocks, quantify their respective impact on potential leakage and conclude on a way to combine them to achieve minimal leakage (regardless of performance); (2) we propose in Section 5 optimized execution

strategies and algorithms leveraging these building blocks with realistic performance while maintaining bounded data leakage.

4 COUNTERMEASURE BUILDING BLOCKS

This section introduces three security building blocks previously sketched in [10] to control potential data leakage on the set O of objects accessible to the UDF $f = agg \circ cmp$, executed inside a Data task DT_f, through the successive results transmitted to an external App.

4.1 Stateless Data Tasks

Since potential attackers control the code of f, an important lever that can be exploited is data persistency, as keeping a *state* between successive executions maximizes leakage. For instance, in the Attack example (Section 3.3), f maintains a variable *leak* according to previous executions to avoid leaking same data twice. Persistent states can be exploited by f –although executed as Data task DT_f– without hurting the security hypotheses, e.g., in memory or resorting to PDMS database or secure file system.

A first building block is to rely on *stateless* Data tasks (without negative impact on usage as database queries are evaluated independently) with the objective of limiting the leakage rate in successive executions:

Definition 3. Stateless Data task. A stateless Data task is instantiated for the sole purpose of answering a specific function call/query, after which it is terminated and its memory wiped.

Enforcement. On SGX, statelessness can be achieved by destroying the Data task's enclave. It also requires to extend containment (security property *P1*) by preventing variable persistency between executions or direct calls to stable storage (e.g., SGX protected file system library). Obviously, PDMS database access must also be regulated by the Core.

Impact on leakage. A corrupted computation code running as a Stateless Data task may only leverage randomness to maximize the leakage rate. For instance, in the Attack example, at each execution, f_{leak} would select a random fragment of O to produce a *leak* –even if the same query is run twice on the same input–. Considering a uniform leakage function, the probability of producing a new *leak* is proportional to the remaining amount of data –not leaked yet– present in the data task input O. That is, the probability is high (i.e., close to 1) when none or only a few fragments of O have been already leaked, and it slowly decreases to 0 when O has been (nearly) entirely leaked, which increases the necessary executions to leak large amounts of data.

4.2 Deterministic Data Tasks

The stateless property imposes on attackers to (i) employ randomness in selecting data fragments to be leaked in a computation to maximize the leakage, and (ii) choose the leaked fragment necessarily in the current computation input (previous inputs cannot be memorized). To further reduce leakage, we enforce a new restriction for Data tasks, i.e., determinism:

Definition 4. Deterministic Data task. A deterministic Data task necessarily produces the exact same result for the same function code executed on the same input, which precludes leakage accumulation in the case of identical executions (enforcing Def. 1).

Enforcement. Data task containment (security property *P1*) can be leveraged to enforce data task determinism by preventing access to any source of randomness, e.g., system calls to random APIs or timer/date. Virtual random APIs can easily be provided to preserve legitimate uses, e.g., the need for sampling, as long as they are "reproducible", e.g., the random numbers used are forged by the Core using a seed based on the function code f and its input set O, e.g., $seed = hash(f||O)$. The same inputs (i.e., same sets of database objects) must also be made identical between successive Data task execution by the Core (e.g., sorted before being passed to Data tasks). Clearly, to enforce determinism, the Data task must be stateless, as maintaining a state between executions provides a source of randomness to the Data task. Note that another way to enforce determinism is to store the previous results produced by the data tasks for any different input objects set, and reuse the stored result instead of recomputing (see Section 5.1).

Impact on leakage. With deterministic (and stateless) Data tasks, the remaining source of randomness in-between computations is the Data task input (i.e., $O_\sigma \subseteq O$). The attacker has to provide a different selection predicate σ at each computation in hope of maximizing the leakage rate. Hence, the average leakage rate of deterministic Data tasks is upper bounded by that of stateless Data tasks. In theory, the number of different inputs of f being high (up to $2^{|O|}$, the number of subsets of O), an attacker can attain similar Data set leakage with deterministic Data tasks as with stateless ones but at lower leakage rates.

4.3 Decomposed Data Tasks

By changing the selection predicate σ (i.e., as needed to favor rich usage for Apps), attackers may leak new data with each new execution of f, regardless if Data tasks are stateless and deterministic. The attacker could also concentrate leakage (see Def. 2) on a specific object o, by executing f on different input sets but each containing the object o. To mitigate the attack vector represented by the selection predicate σ, we introduce a third building block based on decomposing the execution of $f = agg \circ cmp$ into a set of Data tasks. On the one side a Data task DT^{agg} executes the code of *agg*, and on the other side a set of Data tasks $\{DT_i^{cmp}\}_{i>0}$ executes the code of *cmp* on a partition of the set of authorised objects O, each part P_i of the partition being of maximum cardinality k.

The goal is to limit the Object leakage since information about objects in a given part P_i can only leak into the k results produced by DT_i^{cmp}. This parameter k is called *Leakage factor*, as it determines the number of intermediate results in which information about any given object o can be leaked. An important observation is that to enforce a leakage factor of k, the partitioning of O in parts of size at most k has to be 'static', i.e., independent of the computation input O_σ, so that any object is always processed within the same $k-1$ others objects across executions, such that the stateless deterministic Data task processing that part always produces the same result set. This further restriction for Data tasks is defined as follows:

Definition 5. Decomposed Data tasks. Let $P(O) = \{P_i\}$ be a *static* partition of the set of objects O, authorized to function $f = agg \circ cmp$, such that any part P_i is of maximum cardinality $k > 0$ (k being fixed beforehand, e.g., at install time). A decomposed Data

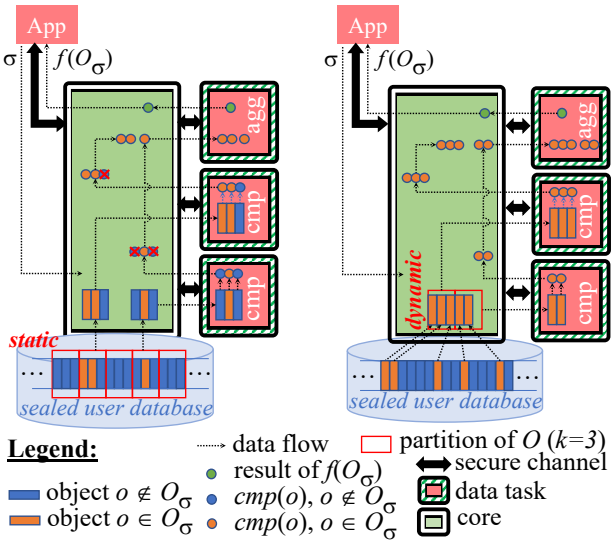


Figure 3: Decomposed (left) or Adaptive (right) execution.

tasks execution of f over a set of objects $O_\sigma \subseteq O$, involves a set of Data tasks $\{DT_i^{cmp} | P_i \cap O_\sigma \neq \emptyset\}$, with each DT_i^{cmp} being a stateless deterministic Data task executing cmp on a given part P_i containing at least one objects of O_σ. Each DT_i^{cmp} is provided P_i as input by the Core and produces a result set $res_i^{cmp} = \{cmp(o), o \in P_i\}$ to the Core. The Core discards all the results corresponding to the objects $o \notin O_\sigma$, i.e., not part of the current computation. A stateless deterministic Data task DT^{agg} is used to aggregate the union of the results sets part of the computation, i.e., $\cup_i \{res_i^{cmp} = \{cmp(o), o \in P_i \cap O_\sigma\}\}$, to produce the final result.

As an illustration, Figure 3 (left) shows a decomposed Data task execution, on a static partition of O with a Leakage factor $k = 3$, evaluating f on 3 objects matching predicate σ (in orange) and present in 2 parts, with 2 Data tasks allocated to evaluate cmp on each part, and one Data task evaluating agg on the result of cmp.

Enforcement. To implement this Decomposed data tasks strategy, it is sufficient to add trusted code to the Core that implements an execution strategy consistent with this definition (Section 5.1 explains this in detail).

Impact on leakage. Any computation involves one or several parts P_i of O. Due to our execution strategy leveraging stateless deterministic Data tasks, the result set $\{cmp(o)\}_{o \in P_i}$ is guaranteed to be unique for any o. Hence, the Data set leakage for any part P_i is bounded by $||cmp|| \cdot |P_i|$, regardless of the number of the successive computations involving any $o \in P_i$. Consequently, the Data set leakage over a very large number n of computations on O is also bounded: ${L_f}^{n \to +\infty}(O) \leq \sum_i ||cmp|| \cdot |P_i| = ||cmp|| \cdot |O|$.

Regarding Object leakage, for any P_i, the attacker has the liberty to choose the distribution of the $|P_i|$ leak fragments among the objects in P_i. At the extreme, all $|P_i|$ fragments can concern a single object in P_i. For any object $o \in P_i$, the Object leakage is thus bounded by ${L_f}^{n \to +\infty}(o \in O) \leq min(||cmp|| \cdot k, ||o||)$, with k the leakage factor equal to the maximum number of objects in any P_i.

Minimal leakage. From above formulas, a decomposed Data task execution of $f = agg \circ cmp$ is optimal in terms of limiting the potential data leakage, with both minimum data set and object leakages, when a maximum degree of decomposition is chosen, i.e., a partition at the object level fixing $k = 1$ as leakage factor.

However, reaching this minimal leakage requires to allocate at runtime one stateless and deterministic Data task per object $o \in O_\sigma$ involved in the computation.

5 PRACTICAL EVALUATION STRATEGIES

The building blocks presented above theoretically allow an evaluation of f with low and bounded leakage, minimal if $k = 1$. However, an evaluation strategy based on a direct implementation may lead to unrealistic performance (see Section 6) in the case of large objects sets, mainly because (i) too many data tasks must be allocated at execution (up to one per object $o \in O_\sigma$ to reach minimal leakage) and (ii) many unnecessary computations are needed (objects $o \notin O_\sigma$ must be processed, given the 'static' partition, if they belong to parts containing objects $o \in O_\sigma$, see Figure 3 (left)).

We need to overcome these obstacles and establish evaluation strategies with acceptable performance in practice, while still maintaining low data leakage. Therefore, we propose new execution strategies leveraging two mechanisms: **Result reuse**, which avoids computations on objects that are not part of the input (objects $o \notin O_\sigma$) and opens the way to new strategies based on a 'dynamic' partitioning of the input objects of f; and **Execution replay**, which allows applying coarser-grained partition schemes with a reduced set of Data tasks allocated at execution, while keeping Object leakage low to minimum.

5.1 Decomposed Execution with Result Reuse

Result reuse consists in storing into the Core any new intermediate result $cmp(o)$ for any object o after its first computation and then reusing this value[2] for all subsequent evaluations of f taking o in input (i.e., $O_\sigma \supset o$).

Result reuse implies processing any raw database object o only once, without the attacker having the opportunity to consider again that object o as input of –potentially corrupted– functions cmp or agg, and thus without any means to further impact data leakage related to o. Hence, 'static' partitioning is not required anymore, and this allows adopting 'dynamic' partition schemes, where the set of –newly computed– objects part of the computation input O_σ can be partitioned and processed independently of other –already computed– objects in $O \setminus O_\sigma$, while still satisfying Def. 5.

This opens to a computation strategy based on (i) a *Generic* execution algorithm with Result reuse which is the common entry point for (ii) a dynamic computation strategy, namely the *Adaptive* execution algorithm presented here, or the *Replay* execution algorithm presented next. All these algorithms (the generic part and the computation strategy algorithms) are considered trusted and are part of the Core, while the codes of agg and cmp are considered untrusted and run therefore as Data tasks.

Generic execution with Result reuse (Algorithm 1). This code module constitutes the generic entry point for any computation of f. It first determines the set of objects O_σ satisfying the

[2]In a PDMS context, we deal mainly with historical data (e.g., electricity traces, GPS histories, medical data, personal images, etc.). An implicit assumption considered in the paper is that the personal database is managed in append only mode (objects are inserted or deleted, but not updated). To extend our proposals to support explicit updates, these can be considered as deletions followed by insertions (see Section 5.3).

Algorithm 1 Generic execution with Result reuse (Core code)

Input: Querier a, public key PK_a, predicate σ defining $O_\sigma \subset O$
Output: Value $v = agg \circ cmp(O_\sigma)$ result of computation

1: $O_\sigma \leftarrow \{o \in O \mid \sigma(o) = true\}$ ▷ objects in query scope
2: $O^+ \leftarrow \{o \in O_\sigma \mid cmp(o) \neq null\}$ ▷ objects with existing cmp
3: $O^- \leftarrow \{o \in O_\sigma \mid cmp(o) = null\}$ ▷ objects with missing cmp
4: $CMP_O^+ \leftarrow \{cmp(o) \mid o \in O^+\}$ ▷ existing $cmp(o)$ values
5: **if** $O^- \neq \emptyset$ **then** ▷ compute missing $cmp(o)$
6: $CMP_O^- \leftarrow$ compute(sort(O^-))
7: store CMP_O^- values in PDMS
8: **end if**
9: $DT^{agg} \leftarrow$ createDT(agg) ▷ create Data Task (agg code)
10: open(DT^{agg}) ▷ open secure channel (attestation)
11: send(DT^{agg}, ($CMP_O^- \cup CMP_O^+$)) ▷ send all cmp
12: $v \leftarrow$ receive(DT^{agg}) ▷ receive the result
13: killDT(DT^{agg}) ▷ kill DT^{agg}
14: **return** $encrypt(v, PK_a)$ ▷ return result encrypted

Algorithm 2 Adaptive execution (Core code)

Input: O^- an ordered set of database objects, k the leakage factor
Output: CMP_O^- a set of $cmp(o)$ values for these objects

1: $m \leftarrow \left\lceil \frac{|O^-|}{k} \right\rceil$ ▷ number of partitions
2: $\{P_i\}_{i \leq m} \leftarrow$ random partitioning of O with $|P_i| \leq k$
3: $CMP_O^- = \emptyset, CPM = \emptyset$
4: **for** i in $(1 : m)$ **do**
5: $DT_i^{cmp} \leftarrow$ createDT(cmp) ▷ create a Data Task
6: open(DT_i^{cmp}) ▷ open secure channel (attestation)
7: send(DT_i^{cmp}, P_i) ▷ send partition
8: $CMP \leftarrow$ receive(DT_i^{cmp}) ▷ receive the results
9: killDT(DT_i^{cmp}) ▷ kill the Data Task
10: $CMP_0^- = CMP_O^- \bigcup CMP$ ▷ add results to resultset
11: **end for**
12: sort CMP_O^- in same order of corresponding objects in O^-
13: **return** CMP_O^-

computation (line 1) and splits it into two sets (lines 2-3): $O^+ \subseteq O_\sigma$ the objects that have already been computed in previous computations of f and $O^- \subseteq O_\sigma$ the objects selected for the first time. Then it constructs CMP_O^+ by retrieving the cmp values stored for the objects in O^+ (line 4). For the objects in O^-, it triggers a *compute* process (line 6) based on the selected computation strategy (i.e., *Adaptive* presented below or *Replay* introduced in Section 5.2), and then stores the results for future use (line 7). Finally, a stateless deterministic Data task DT^{agg} is created to aggregate the entire set of all values in $\{cmp(o), o \in O_\sigma\}$ (lines 9-13) before sending the final result to the App encrypted with its public key PK_a.

Adaptive execution (Algorithm 2). This execution strategy implements the *compute* function (see line 6 in Algorithm 1). It is called *Adaptive* as it leverages the results reuse to perform a 'dynamic' partitioning of O_σ, i.e., computed progressively based on each input of the queries in the workload, as opposed to the 'static' partition method in Section 4.3 (see Figure 3). Given an input O^- of database objects never computed before and a value k indicating a maximum cardinality, it builds a random (randomness being required to avoid the attacker to have knowledge of the objects placed in a same partition) partition $\{P_i, |P_i| \leq k\}_{i \in [1,m]}$ of O^- with $m = \lceil \frac{|O^-|}{k} \rceil$. Then, a set of m stateless deterministic Data tasks is instantiated and each DT_i^{cmp} with $i \in [1, m]$ evaluates the function cmp on part P_i producing a result set $\{cmp(o)\}_{o \in P_i}$. The final result CMP_O^- is the union of all the result sets corresponding to the partition.

Leakage analysis. The analysis detailed in Section 4.3 remains entirely valid for Adaptive strategy. The reason is that due to Result reuse, a unique opportunity per object to leak information is permitted, thus Data set and Object leakages results still hold.

Performance considerations. In terms of performance, Adaptive has two advantages compared to the Decomposed solution proposed in Def. 5: on the one hand, it allows to process only the database objects useful to the computation, i.e., objects $o \in O_\sigma$; on the other hand, the dynamic nature of Adaptive partitioning allows to reduce the number of involved Data tasks to $m = \lceil \frac{|O^-|}{k} \rceil$, with $O^- \subseteq O_\sigma$ the set of newly computed input objects. However, to reach minimal leakage, it is necessary (as for the theoretical solution of Def. 5) to consider singleton partitions (i.e., a leakage factor $k = 1$), which imposes one Data task per newly computed object. Such a large number of Data tasks may proscribe this solution in practice, at least in scenarios (e.g., energy use-case) requiring various computations on large sets of objects, due to the performance overhead of creating enclaves to host the numerous Data tasks, as confirmed in our measurements in Section 6 (and in line with previous studies on data processing in TEEs like [9, 23]).

5.2 Decomposed Execution with Replay

An ultimate solution would enable processing larger sets of objects to reduce the number of Data tasks involved in the execution, while supporting low leakage factor. Therefore, we propose an execution strategy called *Replay*, which relies on the following observation. Consider two sets of objects O_1 and O_2, with one single object o in common, i.e., $O_1 \cap O_2 = \{o\}$, and two deterministic Data tasks DT_1^{cmp} and DT_2^{cmp}, which respectively receive O_1 and O_2 as input and produce the results sets $\{res_1\}_{o \in O_1}$ and $\{res_2\}_{o \in O_2}$ as output. If both Data tasks produced for object o an identical result value (i.e. $res_1[o] \equiv res2[o]$) then this result does not contain information about objects in $(O_1 \cup O_2) \setminus \{o\}$ (or this information is the same).

The principle of Replay execution generalizes the above intuition for evaluating f on a given input set O^-. To guarantee by construction an evaluation with a leakage bounded by a leakage factor k (as in Adaptive), the idea is to partition O^- into m disjoint parts $\{P_1, P_2, ..., P_m\}$ of (approximately) equal (and large) size, and use each Data task DT_i^{cmp} to produce the set of results $\{cmp(o)\}_{o \in P_i}$ for one P_i. This is replayed several times, each time with a different m-partitioning of O. The partitioning is computed such that after R replays, for any object $o \in O^-$, there remains exactly $k = \frac{|O^-|}{m^R} - 1$ common objects in the intersection of the R successive partitions containing o. The value of k corresponds to the Leakage factor indicating the number of results in which malicious code may inject information about given object o. Therefore, a number of replays $R = \lceil log_m(O^-) \rceil$ guarantees a minimum $k = 1$ value, where any

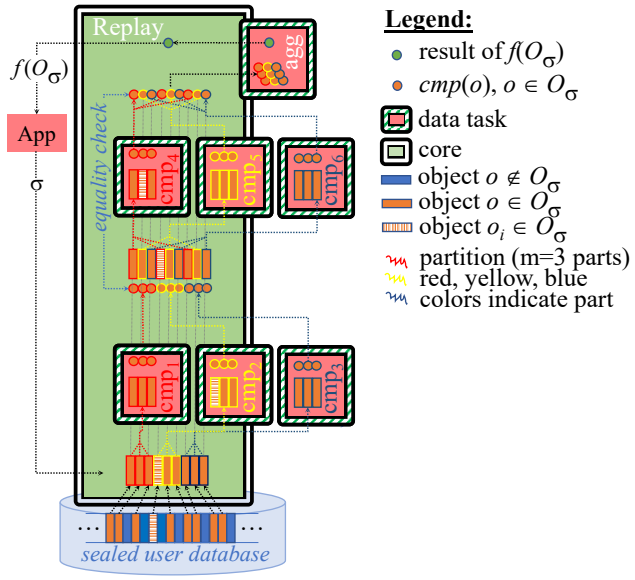


Figure 4: Evaluating f with Replay Algorithm.

object $o \in O^-$ is the only common object in the intersections of the R partitions containing o. The corresponding algorithm, called *Replay*, is illustrated in Figure 4 with $k = 1$, $m = 3$ and a set O^- of 9 objects. Note for example that the hatched object o_i is first included in a yellow part processed by Data task cmp_2, then in a red part processed by cmp_4, and is thus the only object at the intersection of these (yellow and red) parts, so that if the result produced for this object for each part is identical, it depends only on this object. The algorithm works as follows:

Replay execution (Algorithm 3). Given an input objects set O^-, a leakage factor k and a number of partitions m (fixed in practice to 3 or 4 for performance reasons, see Section 6), the number of replay iterations R is computed to reach the expected leakage factor (line 1). Then at each iteration $r \in [1, R]$, the input objects set O^- is partitioned into m parts such that the object ranked in the position j in the input is included within part P_i iff $(\lfloor j \cdot m^r / n \rfloor \mod m) = i$ (line 4). A stateless deterministic Data task DT_i^{cmp} is created for each part and computes cmp on $\{o_j\}_{o_j \in P_i}$. The Core checks that the results $r^*_{o_j}$ obtained for each $o_j \in O^-$ is consistent with the previously computed values r_{o_j} (line 9). Finally, the complete set of results is returned (line 14).

Leakage analysis. Replay execution guarantees by construction that after a number of replays $R = \lceil log_m(n/k) \rceil$ (with $n = |O^-|$), any object $o \in O^-$ was already processed as part of R partitions, and that the intersection of all parts of these partitions containing o is indeed equal to a set of objects containing $\{o\}$ and of cardinality (at most) k. Since all the successive results associated with o for each of these parts was checked to be identical, information about o can only leak into k results. Fixing $k = 1$ guarantees a minimum Object leakage (Def. 2) as in Adaptive. And as Adaptive, the Data set leakage (Def.1) is minimum due to Result reuse, ensuring the uniqueness of $cmp(o)$ for any o regardless of the number of computations including o.

Performance considerations. Compared with Adaptive, Replay involves an increased number of computations (for each object o, $cmp(o)$ is evaluated R times instead of once). Even with minimum

Algorithm 3 Replay execution (Core code)

Input: $O^- = \{o_j\}_{j \in [1,n]}$ an ordered set of n database objects, k the leakage factor, m the number of partitions per replay
Output: CMP_O a set of $cmp(o)$ values for these objects

1: $R \leftarrow \lceil log_m(n/k) \rceil$ ▷ number of replays
2: **for** r in $[1, R]$ **do** ▷ for each replay iteration
3: **for** i in $[1, m]$ **do** ▷ for each part
4: $P_i \leftarrow \{o_j \in O, (\lfloor j \cdot m^r / n \rfloor \mod m) = i\}$
5: $DT_i^{cmp} \leftarrow$ createDT(cmp)
6: send(DT_i^{cmp}, P_i)
7: $\{r^*_{o_j}\}_{o_j \in P_i} \leftarrow$ receive(DT_i^{cmp})
8: killDT(DT_i^{cmp})
9: **if** $\exists o_j \in P_i, r^*_{o_j} \neq r_{o_j}$ (r_{o_j} from previous replays) **then**
10: **return** ERROR
11: **end if**
12: **end for**
13: **end for**
14: **return** $CMP_O^- = \{r_{o_j}\}_{j \in [1,n]}$

leakage factor $k = 1$, the maximum number of Data tasks involved remains acceptable in practice, i.e., $m \cdot \lceil log_m(|O^-|) \rceil$ Data tasks.

5.3 Limitations of the Approach

The security guarantees of our strategies are based on the hypothesis that the Core is able to evaluate the O_σ selection predicates of the App. This is a reasonable assumption if basic predicates are considered over some metadata associated with the objects (e.g., temporal, file/object type or size, tags). However, because of the Core minimality, it is not reasonable to assume the support of more complex selection predicates within the Core (e.g., spatial search, content-based image retrieval). Advanced selection would require specific data indexing and should be implemented as Data tasks, which calls for revisiting the threat model and related solutions.

Another limitation is that our study considers a single *cmp* function for a given App. For Apps requiring several computations, our leakage analysis still applies for each *cmp*, but the total leak can be accumulated across the set of functions. It is also the case if Apps collude. Also, the considered *cmp* do not allow parameters from the App (e.g., *cmp* is a similarity functions for time series or images having also an input parameter sent by the app). Parameters may introduce an additional attack channel allowing the attacker to increase the data leakage.

This study does not discuss data updates. Personal historical data (mails, photos, energy consumption, trips) is append-only (with deletes) and is rarely modified. From the viewpoint of the proposed strategies, an object update can be seen as the deletion and reinsertion of the modified object. An at each reinsertion, the object is exposed to some leakage. Hence, with frequently updated and queried objects, new strategies may be envisioned.

To reduce the potential data leakage, complementary security mechanisms can be employed for some Apps, e.g., imposing a query budget, limiting the σ predicates. Defining such restrictions and incorporating them into App manifests would definitely makes sense, but it is left as future work, as in this paper, we wanted to

	Energy	GPS
Data points number	2 075 259	24 876 978
Objects number	34 587	18 670
Object size (data points - bytes)	60 - 720	1 332 - 31 968
Task for *cmp*	Integral	Length
Result size of *cmp* (bytes)	4	4
Task for *agg*	Average	Sum
Result size of *agg* (bytes)	4	4

Table 1: Considered Data and Query sets

be generic w.r.t. the application types and studied the worst-case scenarios. Also, aggregate computations are generally basic and as such could be computed by the Core. The computation of *agg* by the Core introduces an additional trust assumption which could help to further reduce the potential data leakages.

6 PERFORMANCE EVALUATION

Our experimental evaluation studies the efficiency and scalability of the proposed execution strategies based on a PDMS implementation using Intel SGX, two real data sets and representative computations.

6.1 Experimental Platform

For all experiments, we use a server with an Intel Xeon E-2276G 6-cores @3.8GHz supporting SGX 1-FLC. Out of the 64 GB of RAM available on the server, 128 MB are reserved for Intel SGX with 93.5MB usable by enclaves. It runs Ubuntu 18.04 (kernel 4.15.0-142) with the SGX DCAP driver v1.41. This portrays the scenario where the PDMS would be deployed entirely (i.e. Core and Data tasks) on the Cloud. Then, to grasp the context of a PDMS running on a user device, we also measured performances on a personal computer having an Intel Core i5-9400H @2.5GHz 4-Cores also supporting SGX 1-FLC with 94MB usable by enclaves. The performances on this PC were similar to the ones on the server with an additional computational overtime of approximately 10% due to a slower CPU.

We developped our own PDMS platform using Open Enclave SDK [30] v0.16.1 which aims at facilitating the development of applications for different TEEs. It currently supports Intel SGX and ARM TrustZone and we used the former in our experiments. Besides, the Core data management module is based on SQLite (v3). We presented this platform in [12].

6.2 Data and Query Sets

Data sets. We use two public data sets of personal data (see Table 1). The first [22] contains the electric power consumption of a household minute by minute over a period of four years. Each object is a time series containing the consumption of one hour (i.e., 60 data points). The second [32] data set contains more than 18.000 GPS trajectories from 182 users using different transportation modes (e.g., car, bus, taxi, bike, walk) over more than five years. For scalability reasons, we consider the complete set as if it was generated by a single user. Each trajectory has different spatial and temporal length with 1332 GPS points for one trajectory on average. The Core stores each trajectory as a an object, making each GPS object 44 times larger on average than the electricity objects. For both data sets, each object is associated with the corresponding date interval used by the Core to select the objects required for a computation.

Query sets. We give App the right to request the execution of two UDFs, one for each data set. For the Energy data set, App is able to receive the average of the energy consumption of the user for any time interval(s) provided by App at execution time through σ. This corresponds to the Energy' running example described in Section 1. For the GPS data set, App can request the sum of the length of GPS trajectories, also for any time interval(s). To carry those computations, appropriate Data tasks are used: two *cmp* Data tasks computing either the integral of the energy consumption or the length of the GPS trajectories and two *agg* Data tasks computing either an average or a sum of integers. All Data tasks produce as output an integer of four bytes to preserve precision. Smaller result sizes can be considered depending on the required accuracy of the result, but this would have a negligible impact on performance and is therefore not considered in this section.

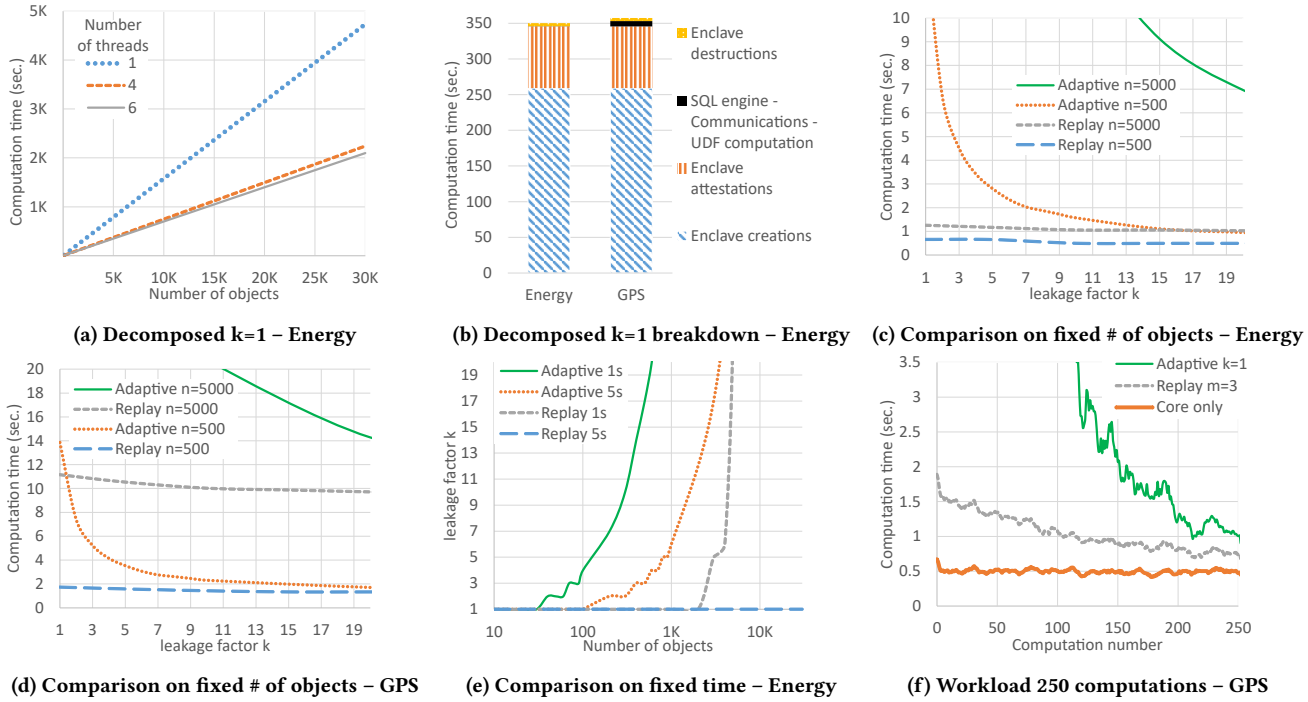
Experimental approach. Our experimental approach consists in three steps. First, we measure the computation times of $agg \circ cmp$ using the Decomposed execution (see Section 4.3) with a maximum degree of decomposition (i.e., leakage factor $k = 1$) over different selectivity (i.e., different σ). Then, we evaluate and compare the trade-off between leakage and performances offered by the Adaptive and the Replay strategies. Finally, we consider the performance of both strategies over a query workload. All reported times are the average of ten computations querying the same number of objects.

6.3 Computation Scalability and Leakage

In this study, we are interested in the trade-off between leakage prevention and performances of the Adaptive and Replay strategies. Particularly, we want to assess the capability of these strategies to reach the minimum leakage factor ($k = 1$) for a computation with limited impact on its performance. As we are not aware of any existing solution for the studied problem to compare it with (see also Section 7), we consider as baseline the Decomposed execution with minimum leakage factor and measure its performance first.

Decomposed with minimum leakage. Figure 5a exposes the computation time with Decomposed execution set up with the minimal leakage $k = 1$ on an increasing numbers of objects from the Energy data set. As expected, its computation time is very high even with a relatively low number of objects (e.g., it exceeds 1 sec. with 13 objects processed) and it increases linearly. Benefiting from the multi-core platform, we measure executions with an optimal number of Data tasks launched in parallel (processed using 6 threads, the number of CPU cores of the machine). Note that even with more CPU cores available, in practice the number of simultaneously running enclaves is limited with SGX depending on enclaves memory footprint [29, p.6]. Despite all our optimization efforts, the computation time remains too high. We obtained similar results with the GPS data set. Unsurprisingly, Decomposed execution is impractical when configured for minimum (or low) leakage factor.

Costs breakdown. Figure 5b details the main execution costs of the Decomposed execution (with $k = 1$) for a computation of 5000 objects. Unsurprisingly, Data tasks creation and attestation represent more than 97% of the execution time. Indeed, on Intel SGX the creation of enclaves incurs a fixed cost depending notably on the number of code pages of the enclave [14]. This is approximately 110ms in our tests. Moreover, the Core has to establish a secure TLS



(a) Decomposed k=1 – Energy

(b) Decomposed k=1 breakdown – Energy

(c) Comparison on fixed # of objects – Energy

(d) Comparison on fixed # of objects – GPS

(e) Comparison on fixed time – Energy

(f) Workload 250 computations – GPS

Figure 5: Performance measurements using Energy and GPS data sets.

channel with each Data task to authenticate it and securely send input data and retrieve results. This takes approximately 40ms per enclave in our tests. These costs are multiplied by the number of Data tasks (one per object in this strategy configuration) hence the high overhead. In comparison, the cost induced by the Core SQL Engine (to compute O_σ), communications and computation of the user-defined functions (all executions of *agg* and *cmp* functions) are marginal. We also note that these data related costs are 22 times higher with the GPS data set compared to Energy, because the GPS objects are larger and thus incur higher data transfer and computation times. Given the cost decomposition, minimizing the number of Data tasks involved in the computation is paramount.

Comparison of execution strategies. Figure 5c and Figure 5d compare the performances of Adaptive and Replay strategies depending on the value of k for both data sets. Increasing the leakage factor k leads to better performance for both strategies because fewer Data tasks are required, but it also increases the Object leakage (increasing the leakage factor of one order of magnitude increases the Object leakage of one order of magnitude, see Section 4). We are hence interested in low to minimum leakage factor values. Two main conclusions can be raised. First, these figures attest to the good performance of Replay, with acceptable execution times around 1 second up to 500 objects and around 10 seconds with 5000 (large) objects. Second, with a small leakage factor (k close to 1), the performance of Replay is around one order of magnitude better than that of Adaptive with both data sets. Indeed, the number of data tasks in the case of a small leakage factor is much lower with Replay (log function in a very high base of the number of objects per part, against a proportional number of the input size with Adaptive). Moreover, although these curves focus on low leakage factors, we highlight the fact that the performance of Adaptive benefits from an increase in the leakage factor (privacy-performance tradeoff), which may lead to preferring the Adaptive strategy to Replay, for example, when processing less sensitive objects, and especially when dealing with large objects. Indeed, the size of the objects has a positive impact on the performance of Adaptive compared to Replay, since the processing cost per object induces a penalty for Replay which requires processing each object several times.

Figure 5e exposes the leakage concentration to be expected on a computation if no more than one (or five) seconds can be spared. Beyond 600 objects, the Adaptive strategy cannot handle a k value lower than 20 under one second. The Replay strategy handles at most 4700 objects with a k value lower than 20 under one second. Within a five second limit, the delivered k value from the Adaptive strategy increases linearly, reaching 20 with only 3400 objects. In opposition, the Replay strategy can handle 30000 objects with the minimal leakage factor under five seconds. For the GPS data set (figure is omitted due to space limitations), the time constraint is more limiting: Adaptive cannot handle more than 1600 objects with a k value lower than 20 under five seconds while Replay can handle up to 2400. Under a time-limit constraint, Replay is thus able to compute more objects with smaller k than Adaptive.

6.4 Execution Costs with Query Workloads

Figure 5f shows the evolution of the computation time over a workload of 250 queries on the GPS data set, with a minimum leakage factor ($k = 1$). The workload is generated so that each computation processes objects belonging to 10 randomly selected time intervals, ranging in size from 2 to 24 hours. 'Core only' corresponds to an execution of the same function f within the Core of the PDMS,

without using any Data task. This represents the case where the security of the UDF code would have been fully verified, and could thus be added to the PDMS Trusted Computing Base (TCB). While this may not be a realistic method for ensuring genericity nor compatible with the considered trust model (see Section 3.2), it allows establishing a lower bound in terms of performance and evaluating the overhead of Data tasks. Replay is again more than one order of magnitude more efficient than Adaptive for the first tens to hundreds queries. Then, the Result reuse strategy benefits to both strategies, and the gap with Adaptive is less significant. On a query workload, Replay is thus the best candidate to ensure minimum leakage while being close to the ideal 'Core only' performance.

7 RELATED WORK

Our solutions are designed to control potential information leakage through the results of successive evaluations of data-driven aggregate functions, whose code is controlled by the recipient of the results but executed at the DBMS side. We focus on the PDMS context, without excluding applications in other contexts, e.g., a traditional DBMS running untrusted UDF code. We hence discuss existing work related to PDMS, DBMS secured with TEE, traditional DBMS addressing issues related to UDF security and finally we position our proposal in the domain of information flow control.

Personal Data Management Systems. PDMSs (also called Personal Clouds, Personal Data Stores, PIMS) are hardware and/or software platforms enabling users to retrieve their personal data (e.g., banking, health, energy consumption, IoT sensors) and exploit them in their own environment.

The user is considered the sole administrator of their PDMS, whether it is hosted remotely within a personal cloud [1–4, 15], or locally on a personal device [13, 15, 16] –solutions such as [15] considering both forms of use–. These solutions usually include support for advanced data processing (e.g., statistical processing, machine learning, on time series or images) by means of applications installed on the user's PDMS plateform [15, 16]. Data security and privacy (which are dominant features of the PDMS) are essentially based on code openness and community audit (by more expert PDMS users) to minimize the risk of misuse leading to data leakage. Automatic network control mechanisms may also help identifying suspicious data transfers [34, 37]. However, no guarantee exists regarding the amount of PDMS data that may be leaked to external parties through seemingly legitimate processed results.

Other PDMS proposals like [6, 8] rely on specific secure hardware (secure microcontroller –or TPM– running a lighweight DBMS engine connected to a mass-storage flash module holding the personal database) as well as a minimal Trusted Computing Base (TCB) for the PDMS, leading to increased security guarantees, but with reduced performance (due to the drastic limitations of the hardware considered) and no possibility to extend the security sphere to untrusted external processing code (which by definition, cannot be included in the TCB). Our approach also relies on secure hardware, but allows the TCB to be coupled with advanced non-secure processing modules to provide control over potential data leakage.

DBMSs secured with TEEs. Many recent works [18, 26, 31, 36, 44] adapt existing database versions to the constraints of TEEs, to enclose the DBMS engine and thus benefit from TEE security properties. For example, Azure SQL [31] allows for encrypted column processing within an enclave, with the cryptographic keys owned client-side being passed to the enclave at runtime, to ensure data confidentiality. EnclaveDB [36] and EdgelessDB [18] embed a –simplified– DBMS engine within an enclave and ensure data and query confidentiality, integrity and freshness. VeriDB provides verifiability –correctness and completeness– of database queries. Finally, [26] introduces a Path ORAM protocol for SGX to avoid data leakage at query execution due to memory access pattern analysis.

These proposals consider the DBMS code running in the enclaves to be trusted by the DBMS owner –or administrator–. Support for untrusted UDF/UADF-like functions is not explicitly mentioned, but would imply the same trust assumption for the UDF code. Our work, on the contrary, makes an assumption of untrusted third-party function code for the database owner, with solutions that apply to classical DBMS context once the P1 security property *Enclaved Core/Data Tasks* (see Section 2.2) can be ensured.

Other proposals [24, 28] combine sandboxing and TEEs to secure data-oriented processing. For example, Ryoan[28] protects the confidentiality of, on the one hand, the source code (intellectual property) of different modules running on multiple sites, and on the other hand, users' data processed by composition of the modules. Despite similarities in the architectural approach, the focus of these works is not on controlling personal data leakage through successive executions and results transmitted to a third party.

Secure UDFs in regular DBMSs. Standard DBMSs support the evaluation of third-party code via user-defined functions (UDF). Products such as Snowflake [5] or Google BigQuery [25] consider the case of "secure" or "authorized" UDFs, where users with UDF execution privilege do not have access rights to the input data. This context is much different from ours because the UDF code is trusted and verified by the administrators with the ability to disable optimizer options (e.g., no selection pushed down the execution tree to avoid revealing certain input data).

Information flow analysis. A body of literature [33, 39, 41] exists in the context of information flow analysis to detect information leakage, but this work is essentially based on the assurance of a property of *non-interference*, which guarantees that if the inputs to a function f are sensitive, no public output of that function can depend on those sensitive inputs. Non-interference is not applicable in our context, since it is legitimate –and an intrinsic goal of running f– for the querier to compute outputs of f that depend on sensitive inputs. Our goal is therefore to quantify, control and limit the potential leakage, without resorting to function semantics. Recent work on code semantic anaylysis [43] considers the case of untrusted third-party applications running in a TEE. They introduce new definitions such as non-reversibility to capture other types of leakage, especially in the context of machine learning algorithms implemented in TEE enclaves. However, the proposal is application specific, it requires access to source code, and the verification code would itself have to be part of the trusted code base, which is considered impractical in our context.

8 CONCLUSION

This paper presents new solutions to mitigate data leakage from untrusted user-defined functions in the PDMS context, satisfying

multiple usage scenarios. Leveraging the emergence of TEEs, we propose security blocks, show the data leakage upper bound, and propose practical implementation strategies that guarantee minimal leakage with reasonable overhead. The solutions are validated on a prototype PDMS [12] using real data sets.

Our solution applies to user-defined aggregate calculation functions useful in many use cases. Perspectives are related to the study of the limitations presented in section 5.3. Another exciting direction is to extend the proposal to large sets of PDMS users wishing to collectively evaluate statistics with minimal leakage and an equitable distribution of data leakage risk among contributors.

REFERENCES

[1] 2007. mydex. https://mydex.org/
[2] 2009. Digi.me. https://digi.me
[3] 2012. BitsAbout.me. https://bitsabout.me
[4] 2017. Personium. https://personium.io/
[5] 2019. Snowflake - Secure UDF. https://docs.snowflake.com/en/sql-reference/udf-secure.html
[6] Tristan Allard, Nicolas Anciaux, Luc Bouganim, Yanli Guo, Lionel Le Folgoc, Benjamin Nguyen, Philippe Pucheral, Indrajit Ray, Indrakshi Ray, and Shaoyi Yin. 2010. Secure personal data servers: a vision paper. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 25–35.
[7] Nicolas Anciaux, Philippe Bonnet, Luc Bouganim, Benjamin Nguyen, Philippe Pucheral, Iulian Sandu Popa, and Guillaume Scerri. 2019. Personal data management systems: The security and functionality standpoint. *Information Systems* 80 (2019), 13–35.
[8] Nicolas Anciaux, Philippe Bonnet, Luc Bouganim, Benjamin Nguyen, Iulian Sandu Popa, and Philippe Pucheral. 2013. Trusted Cells : A Sea Change for Personnal Data Services. In *Proceedings of the 6th biennal Conference on Innovative Database Research (CIDR 2013)*.
[9] Stefan Brenner, Michael Behlendorf, and Rüdiger Kapitza. 2018. Trusted Execution, and the Impact of Security on Performance. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 28–33.
[10] Robin Carpentier, Iulian Sandu Popa, and Nicolas Anciaux. 2021. Poster: Reducing Data Leakage on Personal Data Management Systems. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021*. 716–718.
[11] Robin Carpentier, Iulian Sandu Popa, and Nicolas Anciaux. 2022. Local Personal Data Processing with Third Party Code and Bounded Leakage. In *Proceedings of the 11th International Conference on Data Science, Technology and Applications, DATA 2022*.
[12] Robin Carpentier, Floris Thiant, Iulian Sandu Popa, Nicolas Anciaux, and Luc Bouganim. 2022. An Extensive and Secure Personal Data Management System Using SGX. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022*. 2:570–2:573.
[13] Amir Chaudhry, Jon Crowcroft, Heidi Howard, Anil Madhavapeddy, Richard Mortier, Hamed Haddadi, and Derek McAuley. 2015. Personal Data: Thinking Inside the Box. *Aarhus Series on Human Centered Computing* 1, 1 (2015).
[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016), 86.
[15] Cozy. 2012. Cozy Cloud. https://cozy.io/
[16] Yves-Alexandre de Montjoye, Erez Shmueli, Samuel S. Wang, and Alex Sandy Pentland. 2014. openPDS: Protecting the Privacy of Metadata through SafeAnswers. *PLoS ONE* 9, 7 (2014).
[17] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Proceedings, Part II*, Vol. 4052. 1–12.
[18] Edgeless Systems. 2021. EdgelessDB. https://www.edgeless.systems/products/edgelessdb/
[19] Taher ElGamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on Information Theory* 31, 4 (1985), 469–472.
[20] European Council. 2016. Regulation EU 2016/679 of the European Parliament and of the Council. *Official Journal of the European Union (OJ)* 59, 1-88 (2016), 294.
[21] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph. D. Dissertation.
[22] Georges Hebrail and Alice Berard. 2012. Individual household electric power consumption Data Set. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption
[23] Anders T. Gjerdrum, Robert Pettersen, Håvard D. Johansen, and Dag Johansen. 2017. Performance of Trusted Computing in Cloud Infrastructures with Intel SGX. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science, CLOSER 2017*. 668–675.
[24] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. 123–135.
[25] Google. 2011. Google BigQuery - Authorized Functions. https://cloud.google.com/bigquery/docs/authorized-functions
[26] Ziyang Han and Haibo Hu. 2021. ProDB: A memory-secure database using hardware enclave and practical oblivious RAM. *Information Systems* 96 (2021).
[27] T. Hardjono, D.L. Shrier, and A. Pentland. 2019. *Trusted Data, revised and expanded edition: A New Framework for Identity and Data Sharing*. MIT Press.
[28] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2018. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–32.
[29] Intel. 2021. Intel SGX for Linux OS v2.15 - Developer Reference.
[30] Microsoft. 2017. Open Enclave SDK. https://openenclave.io
[31] Microsoft. 2019. Azure SQL - Always encrypted with secure enclaves. https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-enclaves
[32] Microsoft Research Asia. 2016. GeoLife GPS Trajectories. https://www.microsoft.com/en-us/download/details.aspx?id=52367
[33] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. 228–241.
[34] E. Novak, P. T. Aung, and T. Do. 2020. VPN+ Towards Detection and Remediation of Information Leakage on Smartphones. In *2020 21st IEEE International Conference on Mobile Data Management (MDM)*. 39–48.
[35] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.
[36] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy, SP 2018*. 264–278.
[37] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. 361–374.
[38] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010*. 297–312.
[39] A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.
[40] A.V. Sambra, E. Mansour, S. Hawke, M. Zereba, N. Greco, A. Ghanem, D. Zagidulin, A. Aboulnaga, and T. Berners-Lee. 2016. Solid: A platform for decentralized social applications based on linked data.
[41] Mingshen Sun, Tao Wei, and John C.S. Lui. 2016. TaintART: A Practical Multi-Level Information-Flow Tracking System for Android RunTime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 331–342.
[42] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 353–366.
[43] Ruide Zhang, Ning Zhang, Assad Moini, Wenjing Lou, and Y Thomas Hou. 2020. PrivacyScope: Automatic Analysis of Private Data Leakage in TEE-Protected Applications. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 34–44.
[44] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-Based Verifiable Database. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. 2182–2194.