Enabling secure data-driven applications: an approach to personal data management using trusted execution environments

Robin Carpentier, Iulian Sandu Popa, Nicolas Anciaux

Enabling Secure Data-Driven Applications: An Approach to Personal Data Management using Trusted Execution Environments

Robin Carpentier[1*], Iulian Sandu Popa[2*] and Nicolas Anciaux[3,4*]

[1]Macquarie University, NSW, 2109, Australia.
[2]DAVID Lab., Univ. Versailles St-Q.-en-Yvelines, Université Paris Saclay, 45 av. des Etats-Unis, Versailles, 78000, France.
[3]PETSCRAFT project-team, Inria, 1 rue Honoré d'Estienne d'Orves, Palaiseau, 91120, France.
[4]LIFO Lab, INSA-CVL, 88 bd Lahitolle, Bourges, 18000, France.

*Corresponding author(s). E-mail(s): robin.carpentier@mq.edu.au; iulian.sandu-popa@uvsq.fr; nicolas.anciaux@inria.fr;

Abstract

In a rapidly evolving landscape, Personal Data Management Systems (PDMS) provide individuals with the necessary tools to collect, manage and share their personal data. At the same time, the emergence of Trusted Execution Environments (TEEs) offers a way to address the critical challenge of securing user data while fostering a thriving ecosystem of data-driven applications. In this paper, we employ a PDMS architecture leveraging TEEs as a fundamental security foundation. Unlike conventional approaches, our architecture enables extensible data processing by integrating user-defined functions (UDFs), even from untrusted sources. Our focus is on UDFs involving potentially large sets of personal database objects, with a novel proposal to mitigate the potential risk of data leakage. We introduce security building blocks to impose an upper bound on data leakage and investigate the efficiency of several execution strategies considering different scenarios relevant to personal data management. We validate the proposed solutions through an implementation using Intel SGX on real datasets, demonstrating its effectiveness in achieving secure and efficient computations in diverse environments.

Keywords: Personal Data Management Systems, User-Defined Functions, Untrusted Code, Information leakage, Trusted Execution Environments

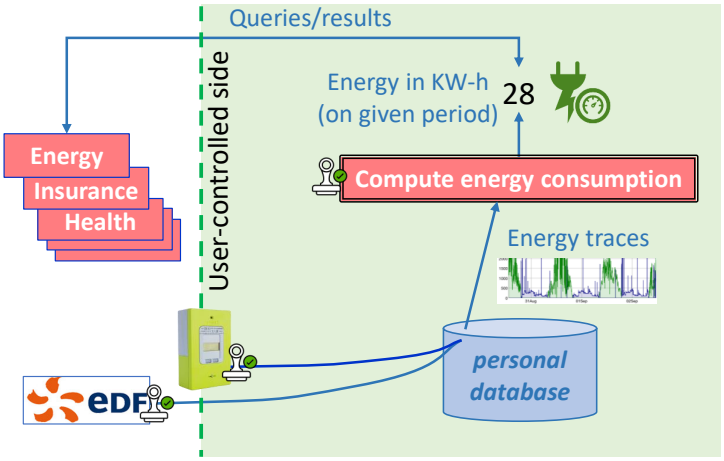


Fig. 1: PDMS computation scenario ('Energy')

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 [1] 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 Cozy Cloud [2], Digi.me [3], Solid [4, 5] and others [6–10], and to large initiatives such as Mydata.org [11] supported by data protection agencies (we refer to [12] for a recent survey).

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.

> ***'Energy' 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 can be easily conceived, 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 inside 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 Apps 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 [13–15] 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 [16, 17]) 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 especially in a context wherein the data of a single user is considered. Similarly, employing secure multiparty cryptographic computation techniques can hurt genericity (e.g., semi-homorphic encryption [18]) or performance (e.g., fully homomorphic encryption [19]), 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 [20] or ARM TrustZone [21], and sandboxing techniques within enclaves, like Ryoan [22] or SGXJail [23]. Our approach consists in considering split execution techniques [24] 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[1] We formalize first 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. Then, we propose a set of execution strategies combining these building blocks to conciliate good performance and information leakage mitigation. Finally, we validate our proposal using representative user-defined computations over two real-world datasets, on an SGX-based PDMS prototype platform. Our extensive experimental evaluation shows that the proposed strategies are efficient for a wide spectrum of PDMS settings which cover well the main dimensions of the studied problem, i.e., the security overhead of TEEs, the communication cost between the components of the PDMS and the size of the computations' input.

This paper is organized as follows. Section 2 formulates the considered problem, by introducing first the main components of our PDMS architecture and the security properties considered, then the computing and threat models. Section 3 presents the basis of our solutions, namely security building blocks, analyzes their impact on information control and identifies an upper bound of information leakage. Section 4 exposes our execution strategies, based on these building blocks, to mitigate information leakage with reasonable performances. To validate our proposal, experiments are conducted in Section 5. Finally, Section 6 describes the related works and Section 7 concludes.

2 Problem Formulation

In this section, we formalize the problem addressed in this paper. First, we introduce the necessary background on Trusted Execution Environments. Then, we describe the Extensive and Secure PDMS architecture that we consider and its associated security properties. We detail our computing model focusing on a common type of data processing on a PDMS, namely *aggregate functions*. We study a data leakage problem, emanating from a powerful attacker programming the data processing code, explained in the subsequent threat model. Finally, we formulate the exact questions that we want to answer and define the required metrics.

2.1 Extensive and Secure PDMS Architecture

2.1.1 Background on TEEs and Related Sandboxing Mechanisms

Trusted Execution Environments (TEEs) represent a promising class of secure hardware which brings security to data storage and data-oriented computations. TEEs achieve security through the creation of a tamper-resistant processing environment [26]. In the last two decades we have witness a proliferation of TEE hardware which is now largely available inside general-public devices such as smartphones and personal computers, and also inside high-end servers. Intel Software Guard Extensions (Intel

[1] This paper is an extension of [24] which introduces the main security building blocks necessary to guarantee a bounded information leakage, and of [25] which introduces a baseline and an efficient execution strategy leveraging the security building blocks. The current paper introduces a new complementary evaluation strategy and an extended experimental evaluation covering the main possible PDMS settings and scenarios. We show that taken together, the proposed strategies allow for secure and efficient computations in all these settings.

SGX) [20, 27] and Arm TrustZone [21, 28] are examples of TEEs embedded inside general-public Intel or Arm CPUs. Their ubiquity and proper computing power make them prime candidates for personal data management systems.

From the security viewpoint, TEEs ensure two properties, i.e., isolation and attestation, which are guaranteed by the hardware, resulting in a high level of confidence. Isolation means that the code running inside the TEE is isolated from the rest of the system, including the operating system and the hypervisor. These trusted environments are commonly called *enclaves* or *secure world* and other programs of the system cannot tamper with the code executing within enclaves. In particular, the data processed by the isolated code is protected from eavesdropping and malicious modification, both from physical and software attacks. This is achieved by encrypting and keeping track of the integrity of the RAM used by the code [29]. While being executed inside the TEE, the isolated code is able to produce a proof of its identity, i.e., attestation, certifying that it is running inside a genuine TEE. Its identity is measured by a cryptographic hash of the memory pages of its code [30, 31]. This proof is signed by a secret key loaded inside the CPU when manufactured or remotely afterwards. This mechanism provides a way to verify by a third pary that a computation result was produced by the expected code running within a certain TEE.

TEEs rely on a trust model where the processor's chip package is the Trusted Computing Base [20]. All the other hardware and software components are considered untrusted. An assumption of current TEEs' design is that the code loaded inside the secure environment is considered trusted as well. Indeed, using TEEs is not a silver bullet towards security: a vulnerable code loaded inside a TEE can still be abused to leak sensitive information [32]. Likewise, the code isolation is not both ways: a malicious code loaded inside the trusted environment can leak its potentially confidential data to the untrusted environment, encrypt documents or participate in denial-of-service attacks [33]. This has led to research works such as [22, 23, 34–36] which propose solutions to isolate programs within their enclaves, a technique known as *sandboxing*. In particular, sandboxing software such as Ryoan [22] or SGXJail [23] provide means to restrict enclave operations (bounded address space and filtered syscalls). This way, one can preclude any voluntary data leakage outside the enclave by a malicious function code.

2.1.2 Architecture Outline

Designing a PDMS architecture that offers both security and extensiveness is a significant challenge: security requires trusted code and environment to manipulate data, while extensiveness relies on user-defined and thus potentially untrusted code. The work in [12] proposes a logical (i.e., abstract) three-layers architecture to handle this tension, where *Applications* (Apps) on which no security assumption is 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).

Core. The Core is a secure subsystem that is a Trusted Computing Base (TCB). It constitutes the sole entry/exit point to manipulate PDMS data and retrieve results,

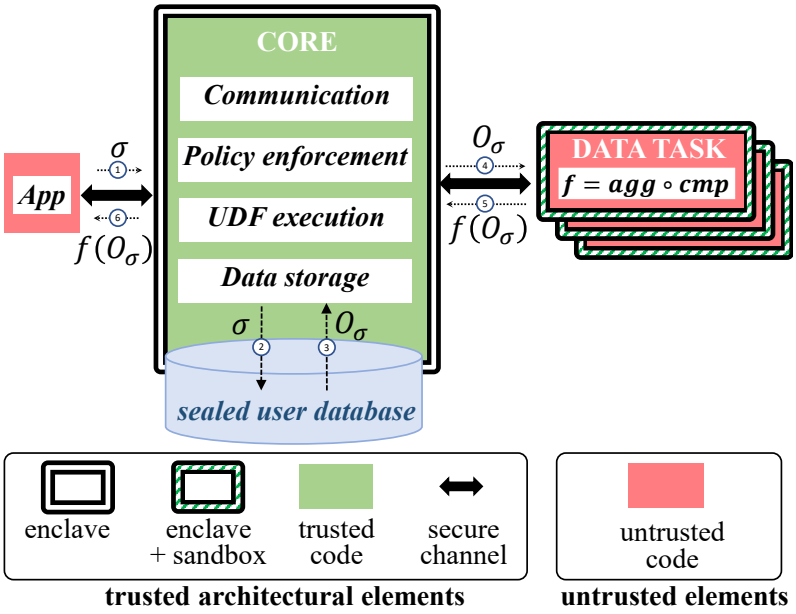


Fig. 2: Architecture overview.

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, a basic query module (as needed to evaluate simple selection predicates on database objects' metadata) and a communication manager for securing data exchanges between the architecture layers and with the Apps. As a TCB, the Core's capabilities should be cut down at the essentials to limit the potential vulnerabilities and make audits easier.

Data tasks. Data tasks can execute arbitrary, application-specific data management code, thus enabling extensiveness (like UDFs in traditional DBMSs). The idea is to handle UDFs 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 applications (large code base) and their execution environment (e.g., web browser) make them vulnerable and potentially malicious. Therefore, no security assumption is made on them. They have no privilege on raw data and only manipulate data resulting from Data tasks and authorized by the Core.

2.1.3 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. Each App requiring the use of UDFs leverages an App Manifest

which includes essential information about the UDFs it needs. This App Manifest should contain:

- The purpose of the UDFs.
- The set of PDMS objects needed to be accessed by the UDFs.
- The size of the result produced by each UDF.
- The identities (i.e., code measurements) of the Data Tasks carrying out the UDF's computation.
- For UDFs requiring the use of multiple Data Tasks, the order in which they should be executed.

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, 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 and the integrity of the data manipulated as well as the integrity of the code execution against the untrusted execution environment (application stack, operating system). Such properties are provided by TEEs, e.g., Intel SGX, as discussed in Section 2.1.1. Besides, the code of each Data task is sandboxed within its enclave to preclude any voluntary data leakage outside the enclave by a malicious Data task function code.
P2. Secure communications. All the communications between the Core, the Data tasks and the Apps are classically secured using TLS to ensure authenticity, integrity and confidentiality. Because the inter-enclave communication channels are secure and attested, the Core can guarantee to the Apps 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. 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' example, a Data task running a UDF computing the energy consumption during a certain time interval will only be supplied by the Core with the necessary time series (i.e., 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 them. Finally, the App only receives final computation results from the Core (e.g., total energy consumption) without being able to access any other data.

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.

2.1.4 Logical versus Physical Architecture

An important observation is that the three layers of the logical architecture introduced above can be deployed within different configurations depending on the use cases which leads to different physical architectures. Let us take a closer look at the Core and Data Tasks which constitute the main elements of our architecture. The straightforward implementation would be to run the Core and the Data Tasks on the same machine,

i.e., a PDMS instance can run on a user-device or on a remote cloud infrastructure (see Section 6 for an overview of existing solutions). However, other cases wherein the Core and the Data tasks run on separated machines can be envisioned. For example, a user of a self-hosted PDMS may need to rely on remote Data Tasks hosted on a powerful server if its own device is not powerful enough to handle some tasks, or because the code of the UDF is protected by intellectual property [22]. Also, in a cloud-based PDMS instance, the Core and the Data Tasks may also be separated, e.g., the Core runs continuously on some cloud machine whereas the Data Tasks are instantiated on demand on some other machines according to the utilization of the cloud resources.

Note that the physical architecture of a PDMS has no impact on its security as long as the security properties defined in Section 2.1.3 hold. However, this has an impact on performance. In particular, a configuration wherein the Core and the Data Tasks run on different machines involves an important communication latency for the data exchanges between the two components. In such cases, appropriate measures have to be taken to keep this overhead at a minimum. Finally, the App can be co-located or not with the other components but this has only a minor impact on performance since the data exchanges between the App and the Core are small, i.e., queries and computation results.

2.2 Computation and Threat Models

2.2.1 Computing 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 in Section 1). 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 (i.e., a function f) as follows: (1) f has legitimate access to large sets of user 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 the PDMS context, e.g., leveraging user GPS traces for statistics of physical activities, the traveled distance or the used modes of transportation over given time periods.

Aggregates in a PDMS are generally applied on complex objects (e.g., time series, GPS traces, documents, images), which require adapted and advanced UDFs at the object level. Specifically, to evaluate an *aggregate* function *agg*, a first *computation* function *cmp* processes 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 denoted by $||cmp||$ 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 this paper on a single App a and computation function f.

Figure 2 shows the main steps of a computation. First, the App requests the execution of f to the Core including a predicate σ. Second, the Core applies σ to select a subset O_σ of the data authorized to the App according to its Manifest (previously validated). Then, the Core demands the creation of a Data Task containing the code of f and establishes a secure communication channel with it, simultaneously verifying its identity. Afterwards, the Core sends the input set O_σ to the Data Task for computation and receives the result $f(O_\sigma)$. Finally, it forwards the result to the App.

Computing model. An App 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 when the App is installed. a can freely invoke f on any $O_\sigma \subseteq O$, where σ is a selection predicate on objects metadata (e.g., a time interval) chosen at query time by a. The function f computes $agg(\{cmp(o) \mid o \in O_\sigma\})$, with cmp an arbitrarily complex preprocessing applied on each object $o \in O_\sigma$ and agg an aggregate function. We consider that agg and cmp are deterministic functions and produce fixed-size results.

2.2.2 Threat Model

We consider that the attacker cannot influence the consent of the PDMS user, required to install UDFs. However, neither the UDF code nor the results it produces can be guaranteed to meet the user's consented purpose. To cover the most problematic cases 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 $O_\sigma \subseteq O$, the input 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 being deterministic and producing the expected results, the execution of f produces some information coveted by the attacker, in order to reconstruct subsets of raw database objects used as input.

On the contrary, we assume that security properties P1 to P3 (see Section 2.1.3) are enforced. In particular, we assume that the PDMS Core code is fully trusted as well as the security provided by the TEE (e.g., Intel SGX) and the in-enclave sandboxing technique. Figure 2 illustrates the trust assumptions on each of the architectural elements of the PDMS involved in the evaluation.

Note that to foster usage, we do not impose any restrictions on the σ predicates nor on the App query budget. In addition, we do not consider any semantic analysis or auditing of the code of f since this is not realistic in our context as the layman PDMS user cannot handle such tasks and the Core, being our Trusted Computing Base, should be kept minimal. These solutions are also mostly complementary to our work as discussed in Section 6.

2.3 Problem Formulation

2.3.1 Research Questions

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.

An analytical study 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 metrics, let us consider a simple attack example.

2.3.2 Attack Example

> ***Attack example.*** The code for f, instead of the expected purpose which users consent to (e.g., computing the energy consumption for the 'Energy' scenario), 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 object 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 the 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 using the above attack example could thus reconstruct O entirely by assembling the received $leak$ after $n = \frac{||O||}{||f||}$ successive executions, with $||O||$ the size in bits needed to encode the information of O.

2.3.3 Leakage Metrics

To address the formulated problem, we introduce two metrics to quantify the amount of data received by the attacker. We denote by $||x||$ the amount of information in x, measured by the number of bits needed to encode it. For simplicity, if x is a function, we consider this value as the size in bits of the result produced by this function. If x is a database object or set of objects, we consider its footprint in bits. We define the leakage $L_f(O)$ resulting from successive executions of a function f on subsets of O allowed to f, as follows:

Definition 1 (Data set leakage)**.** The successive executions of 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 non identical executions of f. Two executions are considered identical if they produce the same result on the same input (it is the case for functions assumed deterministic). Thus, n successive non-identical executions can generate up to a *Data set leakage* of size ${L_f}^n(O) = ||f|| \times n$ (i.e., each execution of f provides up to $||f||$ new bits of information about O).

To quantify the number of executions 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 the 'qualitative' aspect of an attack, we introduce a second leakage metric:

Definition 2 (Object leakage)**.** For a given 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.

2.3.4 Problem Statement

Securing the evaluation of a function which is potentially untrusted from the PDMS user's point of view and has access to a large volume of the user's personal data, leads to the following dual problem statement: (i) design security execution rules that limit the potential data leakage according to the defined metrics to low values, even if the function is executed a large number of times, and (ii) propose efficient evaluation plans that can be used in practice.

To this end, we devise two important steps in the following sections. First, we propose in Section 3 three security design rules to be enforced during the computation of f to limit the leakage of data as measured by the leakage metrics defined above. Named *security building blocks*, they enable the definition of an upper bound on leakage which can be brought down to a minimum. Second, we build upon these design rules and provide algorithms, named *execution strategies*, to efficiently enforce this upper bound Section 4.

3 Leakage Control - Security Building Blocks

This section introduces three security building blocks previously sketched in [37] to control the potential data leakage occurring over successive computation results produced by a UDF $f = agg \circ cmp$, executed inside a Data task DT^f, on a set of objects O. These building blocks allow the establishment of an upper-bound on the amount of data that can be leaked through the results of computations.

3.1 Stateless Data Tasks

Since the potential attacker controls 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 2.3.2), 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., by leveraging in-memory storage or resorting to PDMS database or secure file system.

A first building block to mitigate this lever is to rely on *stateless* Data tasks with the objective of limiting the leakage rate in successive executions. This does not negatively impact usage as computation requests should be evaluated independently, i.e., without being influenced by past or concurrent computations, comparable to database queries in standard DBMSs.

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 accesses must also be regulated by the Core.

Impact on leakage. Without the possibility to rely on a persistent state, a corrupted computation code running as a Stateless Data task may leverage randomness to maximize the leakage rate. For instance, in the Attack example, at each execution, f_{leak} can select a random fragment of O to produce a *leak*, the randomness being drawn from APIs or timer/date. Even if the same query is run twice on the same input, two different fragments of O can be leaked. 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.

3.2 Deterministic Data Tasks

The stateless property forces 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 and tackle attacks relying on randomness, 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 (see Def. 1) in the case of identical executions.

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 (further discussed in Section 4.2).

Impact on leakage. With deterministic (and stateless) Data tasks, the only 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 for each computation in hope of maximizing the leakage rate. Hence, the 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.

3.3 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, each one 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 hand a Data task DT^{agg} executes the code of agg, and on the other hand a set of Data tasks $\{DT_i^{cmp} \mid i \in [1, m]\}$ executes the code of cmp on the partitioned set O of objects authorised to App, each partition P_i being of maximum cardinality k. Each Data Task DT_i^{cmp} produces one result per object in its input partition P_i.

The goal of this decomposition is to limit the Object leakage since information about objects in a given partition P_i can only leak into the (at most) 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 partitions 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 $k-1$ other objects across executions, such that the stateless and deterministic Data task processing this partition always produces the same result set. This additional restriction for Data tasks is defined as follows:

Definition 5 (Decomposed Data tasks)**.** Let $P(O) = \{P_i \mid i \in [1, m]\}$ with $m = \left\lceil \frac{|O|}{k} \right\rceil$ be a *static* partitioning of the set of objects O, authorized to function $f =$

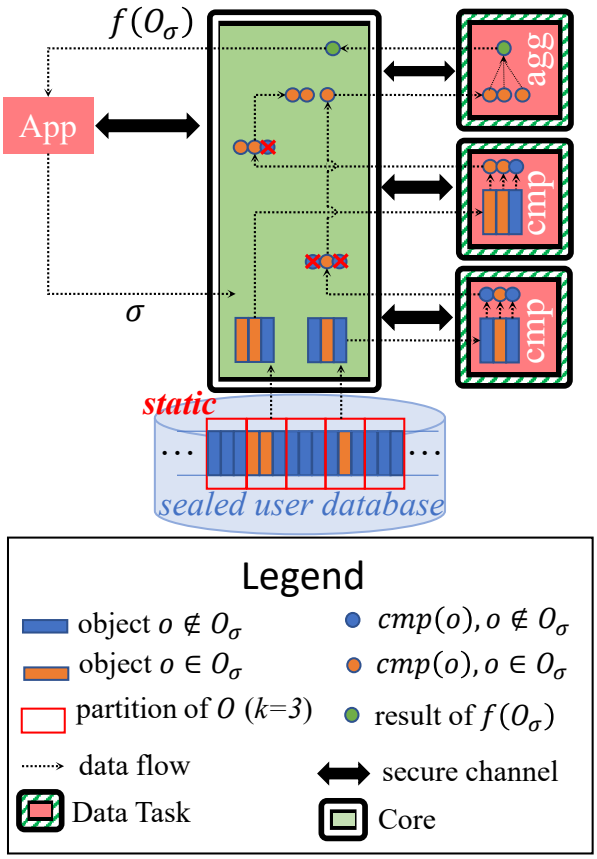


Fig. 3: Decomposed execution with static partitioning.

$agg \circ cmp$, such that every partition P_i is of maximum cardinality $k > 0$ (k being fixed beforehand, e.g., at install time). A decomposed Data tasks execution of f over a set of objects $O_\sigma \subseteq O$, involves a set of Data tasks $\{DT_i^{cmp} \mid P_i \cap O_\sigma \neq \emptyset\}$, with each DT_i^{cmp} being a stateless and deterministic Data task executing cmp on a given partition P_i containing at least one object of O_σ. Each DT_i^{cmp} is provided P_i as input by the Core and produces a result set $res_i^{cmp} = \{cmp(o) \mid o \in P_i\}$ to the Core. The Core discards all the results corresponding to the objects $o \notin O_\sigma$, i.e., objects that are not part of the current computation but have been computed nonetheless since they belong to partitions containing objects that *are* part of the current computation. Ultimately, a stateless and deterministic Data task DT^{agg} is used to aggregate the union of the results sets part of the computation, i.e., $\{cmp(o) \mid o \in O_\sigma\}$, to produce the final result.

As an illustration, Figure 3 shows a decomposed Data task execution, on a static partitioning of O with a Leakage factor $k = 3$, evaluating f on three objects (in orange)

matching predicate σ and present in two partitions, with two Data tasks allocated to evaluate *cmp* on each partition, 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 4.2 explains this in detail).

Impact on leakage. Any computation involves one or several partitions P_i of O. Due to our execution strategy leveraging stateless and deterministic Data tasks, evaluating *cmp* on a specific partition P_i is guaranteed to always produce the same result set $\{cmp(o) \mid o \in P_i\}$ and thus the same result $cmp(o)$ for any $o \in O$.

Hence, the Data set leakage for any partition P_i is bounded by $||cmp|| \cdot |P_i|$, regardless of the number of 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 which fragment of P_i to put inside each of the $|P_i|$ results produced by the Data Task DT_i^{cmp}. 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.

3.4 *Energy* Scenario Leakage Analysis

As an example, we illustrate the impact on leakage with the *Energy* scenario. As in our experimental evaluation (see Section 5), let use consider that each database object is a time series covering the electric power consumption each minute for 1 hour with each data point requiring 12 bytes. Hence, $||o|| = 60 \times 12 \times 8 = 5760$ (i.e., each energy trace is encoded with 5760 bits of information). Let us assume that $||cmp|| = 32$ (i.e., 32 bits to indicate indicate the total consumption of a trace) and $||agg|| = 32$ (i.e., 32 bits to average the consumption over some time period).

Without any building block enforced on the execution of f, an attacker needs 180 queries (i.e., 5760/32) to obtain an object o. By using a stateless Data Task for f, the number of queries to obtain o is (much) higher due to random leakage and o has to be contained in the input of each query. With a stateless and deterministic Data Task for f, the number of queries to obtain o is at least the same as with a stateless f but each query has to have a different input while still containing o. Finally, with a fully decomposed execution of f (i.e., $k = 1$) using stateless and deterministic Data Tasks, only $||cmp|| = 32$ bit of o can be leaked (i.e., less than 3 data points among the 60 points in the time series) regardless of the number of queries.

3.5 Conclusion

The above presented security building blocks hinder the ability of Data Tasks to leak efficiently and control a potentially unlimited leakage of data across computations, by preventing them from relying on persistent state, randomness and the complete view of the input data. These security building blocks greatly improve the capacity of our architecture to protect against leakage and impose, taken together, an upper bound

on the amount of data that can be obtained by an attacker even for an unlimited number of computations.

From the above formulas of this upper bound, a decomposed execution of $f = agg \circ cmp$ using stateless and deterministic Data Tasks 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 partitioning at the object level by 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, which has a negative impact on performance.

4 Leakage Control - Efficient Evaluation Strategies

This section is devoted to the second part of our problem (see Section 2.3.4), i.e., addressing the main issue arising with a direct implementation of our security building blocks: performance. First, we describe two mechanisms aiming at improving the performance while maintaining the low data leakage offered by the building blocks. Then, we present three execution strategies leveraging both the building blocks and these mechanisms to run computations efficiently while guaranteeing low data leakage.

4.1 Performance Overhead

The building blocks presented in the previous section enable 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 5) in the case of large sets of objects, mainly because (i) many unnecessary computations are needed (objects $o \notin O_\sigma$ must be processed, given the 'static' partitioning, if they belong to partitions containing objects $o \in O_\sigma$, see Figure 3), and (ii) too many data tasks must be allocated at execution (up to one per object $o \in O_\sigma$ to reach minimal leakage).

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:

1. **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;
2. **Execution replay**, which allows applying coarser-grained partitioning schemes with a reduced set of Data tasks allocated during the execution, while keeping object leakage low to a minimum.

4.2 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., $o \in O_\sigma$).

[2]In a PDMS context, we deal mainly with historical data (e.g., electricity or GPS traces, medical data, personal images, etc.). An implicit assumption considered in this paper is that the personal database is managed in append only mode: objects are inserted or deleted, but not updated (see Section 4.4).

Result reuse implies processing any raw database object o only once in its lifetime, without the attacker having the opportunity to consider again that object o as input of the –potentially corrupted– function *cmp*, 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' partitioning 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 relying on a decomposed execution (Def. 5).

This paves the way for 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 hereafter, and the *Reverse-and-Replay* and the *Repartition-and-Replay* execution algorithms presented next. All these 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.

4.2.1 Generic Execution with Result Reuse (Algorithm 1)

Algorithm 1 Generic execution with Result reuse (Core code)

Input: Querier a, predicate σ defining $O_\sigma \subseteq O$
Output: Value $v = agg\,(cmp(O_\sigma))$ 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* values
3: $O^- \leftarrow \{o \in O_\sigma \mid cmp(o) = null\}$ ▷ objects with missing *cmp* values
4: $CMP_{O^+} \leftarrow \{cmp(o) \mid o \in O^+\}$ ▷ existing $cmp(o)$ values
5: **if** $O^- \neq \emptyset$ **then** ▷ compute missing *cmp* values
6: $CMP_{O^-} \leftarrow$ compute(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$(DT^{agg}$, sort $(CMP_{O^-} \bigcup CMP_{O^+}))$ ▷ send *cmp* values
12: $v \leftarrow$ receive(DT^{agg}) ▷ receive the result
13: killDT(DT^{agg}) ▷ kill DT^{agg}
14: **return** v ▷ return result

This code module run by the Core constitutes the generic entry point for any computation of f. It first determines the set of objects O_σ satisfying the 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, *Reverse-and-Replay* introduced in Section 4.3.2 or *Repartition-and-Replay* exposed in Section 4.3.3), and then stores the results for future use (line 7). Finally, a stateless

and deterministic Data task DT^{agg} is instantiated to aggregate the entire set of all values $\{cmp(o) \mid o \in O_\sigma\}$ (lines 9-13) before sending the final result to the App.

4.2.2 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' partitioning method in Section 3.3. Figure 4 illustrates the Adaptive strategy. Given an input O^- of database objects never computed before and a value k indicating a maximum cardinality, it builds a partitioning $\{P_i \mid i \in [1,m] \wedge |P_i| \leq k\}$ of O^- with $m = \left\lceil \frac{|O^-|}{k} \right\rceil$. Then, a set of m stateless and deterministic Data tasks is instantiated and each DT_i^{cmp} with $i \in [1,m]$ evaluates the function cmp on the partition P_i and produces a result set $\{cmp(o) \mid o \in P_i\}$. The final result CMP_{O^-} is the union of all the result sets.

Algorithm 2 Adaptive execution (Core code)

Input: O^- a 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: $\{P_i \mid i \in [1,m]\} \leftarrow$ partitioning of O^- with $|P_i| \leq k$
3: $CMP_{O^-} \leftarrow \emptyset, CMP_* \leftarrow \emptyset$
4: **for** i in $[1,m]$ **do**
5: $\quad DT_i^{cmp} \leftarrow$ createDT(cmp)
6: $\quad$ open(DT_i^{cmp})
7: $\quad$ send(DT_i^{cmp}, P_i) ▷ send partition
8: $\quad CMP_* \leftarrow$ receive(DT_i^{cmp}) ▷ receive result set
9: $\quad$ killDT(DT_i^{cmp})
10: $\quad CMP_{O^-} \leftarrow CMP_{O^-} \bigcup CMP_*$
11: **end for**
12: **return** CMP_{O^-}

4.2.3 Leakage and Performance Analyses

Leakage analysis. The analysis detailed in Section 3.3 remains entirely valid for the Adaptive strategy. Now, instead of relying on static partitioning to impose determinism, the reuse of *cmp* results offers only a unique opportunity per object to leak information, thus the conclusions on Data set and Object leakages detailed in that section still hold. Note that the determinism of *agg* still requires sorting (line 11 of Algorithm 1).

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

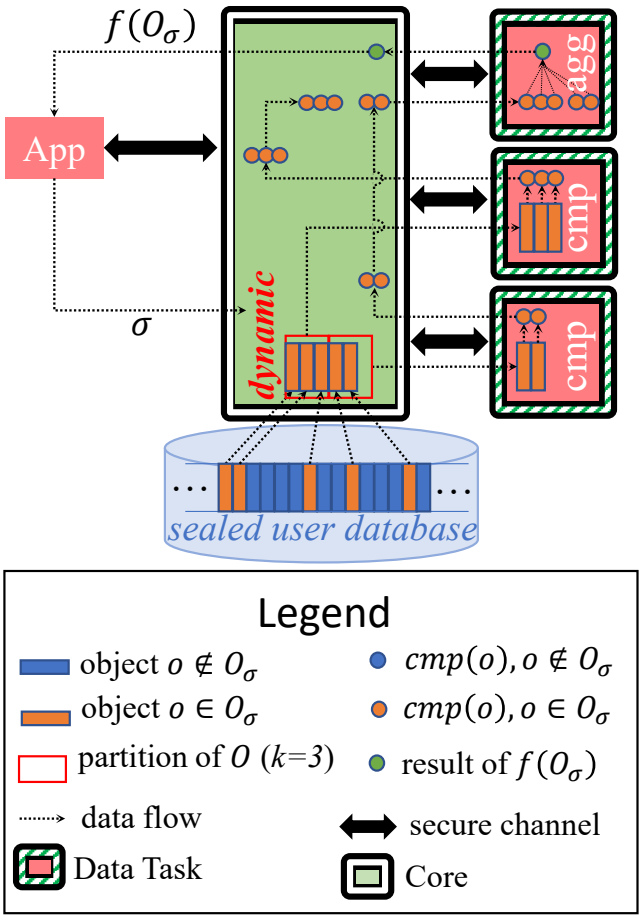


Fig. 4: Evaluating f with Adaptive Algorithm.

$o \in O_\sigma$; on the other hand, because it does not compute objects that have already been computed in the past, it reduces the number of involved Data tasks to $m = \left\lceil \frac{|O^-|}{k} \right\rceil$, 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 5 (and in line with previous studies on data processing in TEEs like [38, 39]).

4.3 Execution Replay

4.3.1 Introduction of the Replay Mechanism

Unlike previous strategies that impose the creation of a Data task for each object in the input to reach minimal leakage during the evaluation of *cmp*, the objective is to propose a solution leveraging execution replay in order to reduce the number of Data tasks involved in the execution while supporting a low leakage factor. *Execution replay* consists in allocating a reduced number of stateless and deterministic Data tasks at runtime, themselves processing the input objects of O_σ several times, the minimality of the leakage being guaranteed by an equality check between the different results produced for the same object.

The general idea is as follows. Consider two sets of objects O_1 and O_2, with one single object o_j in common, i.e., $O_1 \cap O_2 = \{o_j\}$, 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 $CMP_1 = \{cmp(o) \mid o \in O_1\}$ and $CMP_2 = \{cmp(o) \mid o \in O_2\}$ as output. We consider that if both Data tasks produce an identical result value for object o_j (i.e. $CMP_1[o_j] = CMP_2[o_j]$) then this result does not contain information about objects in $(O_1 \cup O_2) \setminus \{o_j\}$.[3]

In the following sections, we propose two strategies leveraging execution replay. The first, *Reverse-and-replay*, computes twice the input set O^- object by object using two Data Tasks, in one order and in the reversed order. The second, *Repartition-and-replay* replays the execution multiple times with different partitioning. Each strategy is efficient with different physical implementations of our PDMS architecture: Reverse-and-replay is suited in a configuration wherein the Core and the Data Tasks are executed on the same machine i.e., communication latency is very low, whereas Repartition-and-replay is fitted whenever remote Data Tasks are used.

4.3.2 Reverse-And-Replay Strategy (Algorithm 3)

Given an input set of objects $O^- \subseteq O_\sigma$, the Reverse-and-replay execution works as follows (see Algorithm 3 and an overview in Figure 5). The Core instantiates two stateless and deterministic data tasks DT_1^{cmp} and DT_2^{cmp} and uses these data tasks in pipeline to compute $cmp(o)$, for all $o \in O^-$, one of the pipeline being reversed from the other (lines 7 to 12). More precisely, the Core provides the input set to DT_1^{cmp} in the sequence order (input: $o_1 \rightarrow o_2 \rightarrow ... \rightarrow o_n$) one object at a time, DT_1^{cmp} producing a result after each input (output: $res_1 = cmp(o_1) \rightarrow res_2 = cmp(o_2) \rightarrow ... \rightarrow res_n = cmp(o_n)$). DT_2^{cmp} processes exactly the same input but in the reverse order (input: $o_n \rightarrow o_{n-1} \rightarrow ... \rightarrow o_1$; output: $res'_n \rightarrow res'_{n-1} \rightarrow ... \rightarrow res'_1$). Finally, the Core checks for results equality, i.e. $\forall j \in [1, n]$, $res_j = res'_j$ (line 16) and, in the positive case, returns the set of computed results.

Leakage analysis. Given the pipelined execution, the results res_j and res'_j produced by the Data tasks for any object o_j can only contain information from objects previously processed by the Data task, i.e., about $\{o_x \mid x \in [1, j]\}$ for res_j and about $\{o_x \mid x \in [j, n]\}$ for res'_j. When these results are identical (i.e., $\forall j$, $res_j = res'_j$), they do not contain information about $\{o_x \mid x \in [1, j[\}$ nor about $\{o_x \mid x \in]j, n]\}$,

[3] This assumption and the related limitations are discussed in Section 4.4.3.

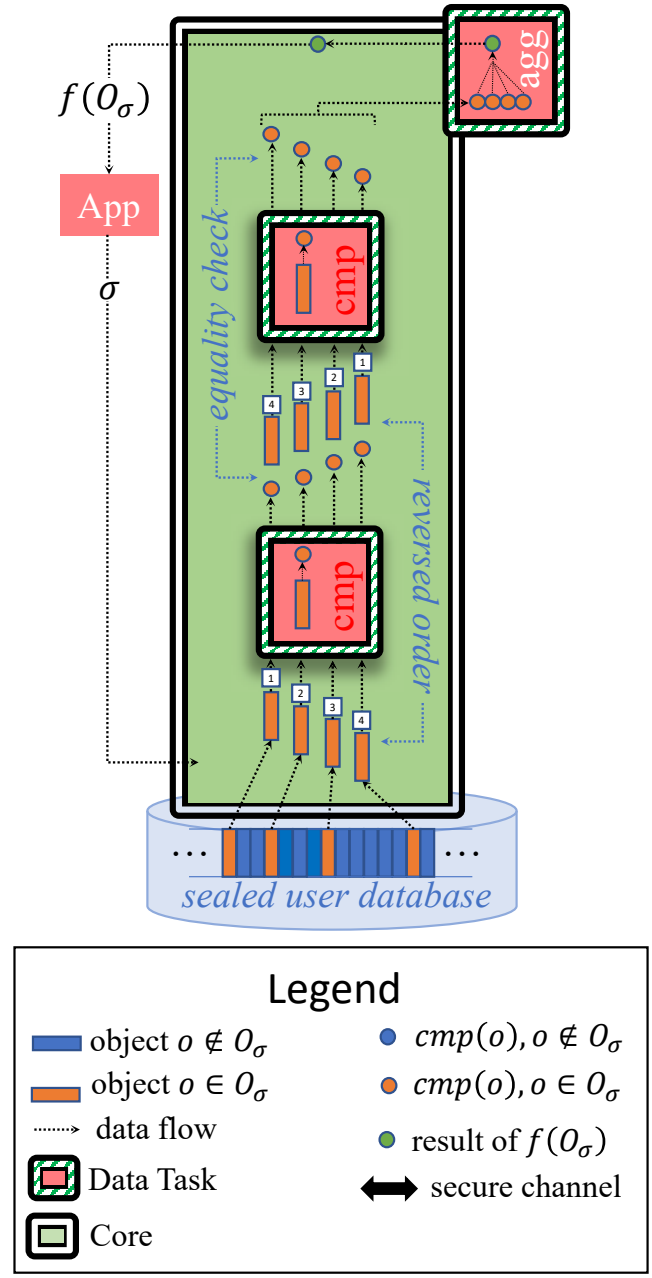


Fig. 5: Evaluating f with Reverse-and-Replay Strategy.

Algorithm 3 Reverse-and-replay execution (Core code)

Input: $O^- = \{o_j \mid j \in [1, n]\}$ a set of n database objects
Output: CMP_{O^-} a set of $cmp(o)$ values for these objects

1: $CMP_1 \leftarrow \emptyset$, $CMP_2 \leftarrow \emptyset$
2: $DT_1^{cmp} \leftarrow$ createDT(cmp)
3: open(DT_1^{cmp})
4: $DT_2^{cmp} \leftarrow$ createDT(cmp)
5: open(DT_2^{cmp})
6: **for** j in $[1, n]$ **do**
7: send(DT_1^{cmp}, o_j) ▷ send the jth object to the 1st DT
8: $res_j \leftarrow$ receive(DT_1^{cmp})
9: $CMP_1 \leftarrow CMP_1 \bigcup \{res_j\}$ ▷ add result to the 1st resultset
10: send(DT_2^{cmp}, o_{n-j+1}) ▷ send the $n-j$th object to the 2nd DT
11: $res'_{n-j+1} \leftarrow$ receive(DT_2^{cmp})
12: $CMP_2 \leftarrow \{res'_{n-j+1}\} \bigcup CMP_2$ ▷ add result to 2nd resultset
13: **end for**
14: killDT(DT_1^{cmp})
15: killDT(DT_2^{cmp})
16: **if** $CMP_1 \neq CMP_2$ **then** ▷ if result sets are not equal
17: **return** ERROR
18: **end if**
19: **return** CMP_1

hence $res_j = res'_j$ only discloses information about o_j. Thus, the Reverse-and-replay execution offers a minimal *object leakage*. Combined with Result reuse's benefits, it guarantees a minimal *data set leakage* too, due to the uniqueness of $cmp(o_j)$ regardless of the number of computations including o_j.

Performance considerations. The main advantage of this strategy is that it requires only two Data Tasks regardless of the number of selected objects for computation. This is much fewer than with the Adaptive strategy, which requires one Data Task per –newly computed– input object to attain minimal leakage. Reverse-and-replay induces an additional cost for computations because every object has to be computed twice, but this has less impact on performance (see Section 5).

However, this strategy does not perform well in all contexts. As stated in Section 2.1.4, one possible physical implementation of our ES-PDMS architecture is to rely on remote Data Tasks, i.e., running on another machine than the one hosting the Core, the two being connected via the Internet. In this scenario, communication latency is several orders of magnitude higher than inter-enclave communication on the same machine. The pipeline mechanism of the Reverse-and-Replay strategy leads to many communications between the Core and the Data Tasks, which heavily suffers from network latency as detailed in Section 5.

4.3.3 Repartition-And-Replay Strategy (Algorithm 4)

Given the performance overhead of the Reverse-and-Replay strategy when dealing with remote Data Tasks, it is important to devise another strategy addressing this weakness. Like Reverse-and-Replay, this strategy should offer better performance than Adaptive when dealing with a low leakage factor, but it should also be robust to communication latency.

The sensitivity to latency of Reverse-and-Replay is due to the fact that each input object causes two communications between the Core and the Data Task: sending the object for computation and receiving its associated result. Reducing the number of communications means that objects have to be sent (and computed) in batch, comparable to the Adaptive strategy with $k > 1$, while still being able to efficiently compute with a low leakage factor.

Based on this idea we designed the following strategy called Repartition-and-Replay. To guarantee by construction an evaluation with a leakage bounded by a leakage factor k (as in Adaptive), the idea is to make a partitioning of the input set O^- into m disjoint partitions $\{P_1, P_2, ..., P_m\}$ of (approximately) equal (and large) size, and use a set of Data tasks $\{DT_i^{cmp} \mid i \in [1, m]\}$, each one producing a set of results $\{cmp(o) \mid o \in P_i\}$ for one P_i. This first step is similar to the Adaptive strategy. However unlike Adaptive, this step is replayed several times, each time with a different m-partitioning of O^-. The partitioning is designed such that after R replays, for any object $o \in O^-$, there remains at most $k = \left\lceil \frac{|O^-|}{m^R} \right\rceil$ 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 a given object o. Therefore, a number of replays $R = \lceil log_m(O^-) \rceil$ guarantees a minimum $k = 1$ value, where any object $o \in O^-$ is the only common object in the intersections of the R partitions containing o.

The corresponding algorithm is illustrated in Figure 6 with $k = 1$, $m = 3$ and a set O^- of nine objects. Note for example that the hatched object o_j is first included in a yellow partition processed by Data task cmp_2, then in a red partition processed by cmp_4, and is thus the only object at the intersection of these (yellow and red) parts, such that if the result produced for this object during the computation of each partition is identical, it depends only on this object. The algorithm works as follows:

Repartition-and-Replay execution. In Algorithm 4, given an input set O^-, a leakage factor k and a number of partitions m, 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 set O^- is partitioned into m partitions such that the object having the index j in O^- is included within partition P_i if and only if $(\lfloor j \cdot m^r / n \rfloor \mod m) = i$ (line 4). A stateless and deterministic Data task DT_i^{cmp} is created for each partition and computes cmp on $\{o_j \mid o_j \in P_i\}$. The Core checks that the results $res^*_{o_j}$ obtained for each $o_j \in O^-$ is consistent with the previously computed values res_{o_j} (line 9). Finally, the complete set of results is returned (line 14).

Leakage analysis. Repartition-and-Replay execution guarantees by construction that after a number of replays $R = \lceil \log_m(n/k) \rceil$ (with $n = |O^-|$), any object $o \in O^-$ has been processed R times as part of R different partitions. Moreover, the intersection

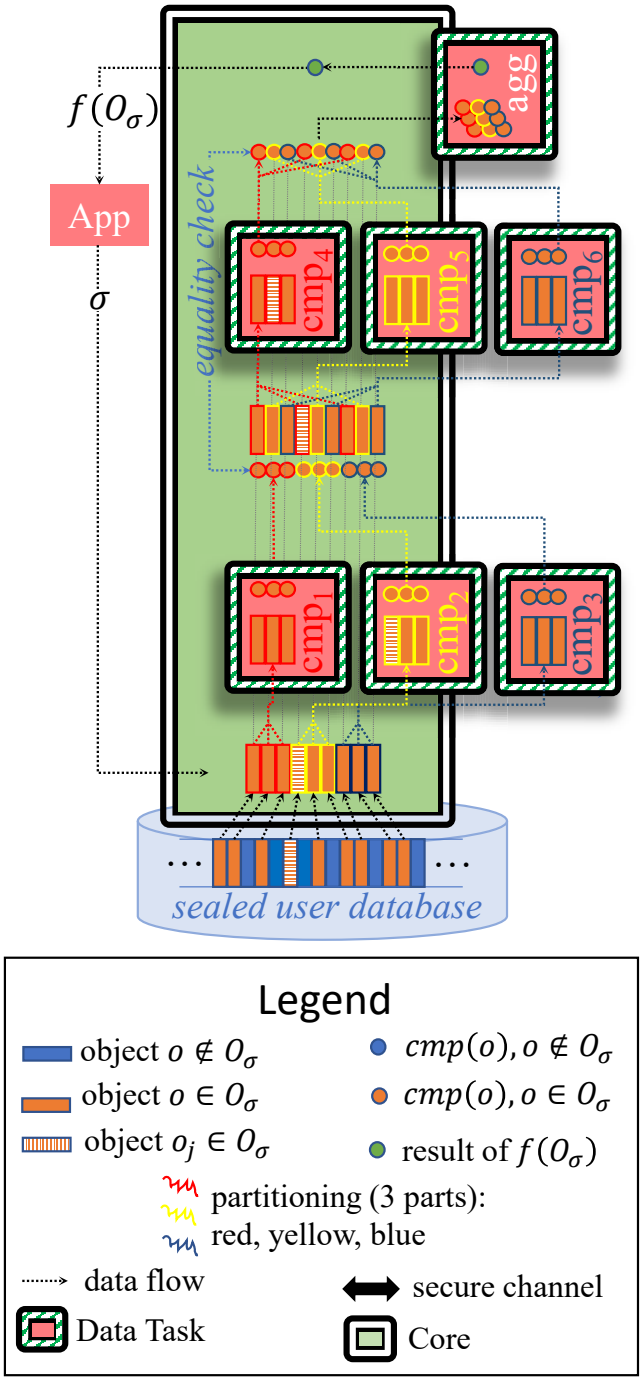


Fig. 6: Evaluating f with Repartition-and-Replay Algorithm.

Algorithm 4 Repartition-and-Replay execution (Core code)

Input: $O^- = \{o_j \mid j \in [1,n]\}$ a 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

$R \leftarrow \lceil \log_m(n/k) \rceil$ ▷ number of replays
for r in $[1, R]$ **do** ▷ for each replay iteration
  for i in [1, m] **do** ▷ for each partition
    $P_i \leftarrow \{o_j \in O^- \mid (\lfloor j \cdot m^r / n \rfloor \mod m) = i\}$
    $DT_i^{cmp} \leftarrow$ createDT(cmp)
    send(DT_i^{cmp}, P_i)
    $\left\{res^*_{o_j} \mid o_j \in P_i\right\} \leftarrow$ receive(DT_i^{cmp})
    killDT(DT_i^{cmp})
    if $\exists o_j \in P_i \mid res^*_{o_j} \neq res_{o_j}$ (res_{o_j} from previous replays) **then**
      return ERROR
    end if
  end for
end for
return $CMP_{O^-} = \left\{res_{o_j} \mid j \in [1,n]\right\}$

of all partitions containing o is indeed equal to a set of objects of cardinality (at most) k. Since all the successive results associated with o for each of these partitions 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, Repartition-and-Replay involves an increased number of computations: for each object o, $cmp(o)$ is evaluated R times instead of once with R being at most $\lceil \log_m(n) \rceil$. However the cost of these re-computations is widely compensated by the reduction in the number of Data Tasks as experimented in Section 5: from at most n Data Tasks with Adaptive, Repartition-and-Replay involves at most $m \cdot \lceil \log_m(n) \rceil$ Data Tasks. Besides, the other strength of this strategy is to overcome the limitation of Reverse-and-Replay with remote Data Tasks. Although it uses $m \cdot \log_m(n)$ Data Tasks instead of 2, Repartition-and-Replay highly reduces the number of communications involved between the Core and the Data Tasks, from $4 \cdot n$ to $2 \cdot m \cdot \log_m(n)$, with m tunable, because inputs are processed by batch. This can have a significant impact on performance when communication latency is significant (see Section 5.3.2).

Tuning the value of m. The number of partitions on each replay iterations, denoted m, has to be chosen beforehand. Note that adjusting m has no impact on the security of the strategy since the same leakage protection can be achieved for any m, with more or less replay iterations. However, m impacts performances as it affects the total number of Data Tasks DT^{cmp} involved in the computation as well as the number of replay iterations, and thus the number of re-computations of each object. Selecting a smaller m leads to fewer DT^{cmp} and subsequently to fewer communications, but

it increases the number of re-computations (as indicated in Table 1 in Section 4.4). For instance, it can be showed that the number of Data Tasks (and communications) is minimized when $m = \lceil e \rceil = 3$ (where e is Euler's number), while the number of computations is minimized when $m = n$ (which is equivalent to the Adaptive strategy with $k = 1$). Our objective is to select the value of m which minimizes the overall execution cost of Repartition-and-Replay depending on different parameters involved such as the number of input objects, the cost of processing one object, the latency etc. To do so, we first approximate the execution time of Repartition-and-Replay as a function of m as follows:

$$C(m) = m \log_m(n) * (\alpha + 2\lambda) + n \log_m(n) * (\beta + \delta + \Omega + \omega + \gamma)$$

where $\alpha, \beta, \delta, \lambda, \Omega, \omega, \gamma$ are constants representing the estimated time in ms for executing different key operations. Specifically, (i) α is the minimal estimated time to create, attest and destroy an enclave regardless of its memory size, β is the additional enclave creation time per object and δ the additional enclave destruction time per object (i.e., the observed enclave creation/destroy has a fixed cost and a variable cost depending on the enclave's memory size); (ii) λ is the observed communication latency between the Core and the Data Tasks, Ω and ω are the transmission time of an object and of a result, respectively, depending on the communication bandwidth and the object/result size; and (iii) γ is the computation time for one object.

By analyzing the derivative of $C(m)$, we can find the value of m minimizing $C(m)$. This leads to:

$$m * (\ln(m) - 1) = \frac{n(\gamma + \Omega + \omega)}{(\alpha + 2\lambda)} \tag{1}$$

with $m \in \mathbb{N}^* \setminus \{1\}$.

4.4 Conclusion, Optimizations and Limitations

In this section, we addressed the efficiency issue arising from a basic application of the security building blocks defined in Section 3. First, we proposed two mechanisms to complete these building blocks by dealing with important slowdown factors:

- **Result Reuse** which avoids unnecessary computations of objects which are not part of the query input.
- **Execution Replay** which reduces the number of Data Tasks required to secure a computation.

Then, we introduced three execution strategies which are execution plans for the considered computations $f = app \circ cmp$ implementing the building blocks and these mechanisms. Table 1 presents the main differences between our strategies for the highest security level (i.e., $k = 1$) and the three major execution costs: the number of Data Tasks, of communications and of computations. It gives an intuition of the strategies' performances which are experimentally validated in Section 5.

- Adaptive has the lowest complexity for the number of communications and computations among the three strategies. However, it requires the highest number of Data

	# of Data Tasks	# of communications	# of computations
Adaptive	n	$2 \cdot n$	n
Reverse-and-replay	2	$4 \cdot n$	$2 \cdot n$
Repartition-and-replay	$m \cdot \log_m(n)$	$2 \cdot m \cdot \log_m(n)$	$n \cdot \log_m(n)$

Table 1: Complexity (depending on $n = |O_\sigma|$) of the main factors impacting the cost of the proposed strategies (for $k = 1$).

Tasks, a factor which may dominate the execution cost of a computation given the high cost of enclave creation.

- Reverse-and-replay trades communication and computation costs (which are doubled compared with Adaptive) for a very low number of Data Tasks (which is independent of the input size). Hence, this strategy is expected to perform best when the enclave creation time dominates the communication/computation costs.
- Repartition-and-replay balances the number of Data Tasks and the communication and computation costs. This strategy has the lowest number of communications. Hence, it is expected to perform best when the communication latency is dominant.

In the remainder of this section we discuss two optimizations allowing to increase the performance of the proposed strategies. Then, we conclude the section with a discussion on the limitations of the proposed solution.

4.4.1 Parallel Data Task Execution

The Decomposed execution security principle (see Section 3.3) leads to a set of Data Tasks DT_i^{cmp} applying each the *cmp* function on a partition of the selected input objects O_σ. Since all these computations are independent, all DT_i^{cmp} can be run in parallel depending on the parallelism capabilities of the system. Hence, a straightforward way to increase the performance of our proposed execution strategies is to execute the *cmp* Data Tasks in parallel. This applies to Adaptive (see Algorithm 2), to Reverse-and-Replay (see Algorithm 3) and to Repartition-and-Replay (see Algorithm 4). Naturally, parallel Data Task execution increases the system resource consumption. In particular, it requires multi-core CPU system capabilities. Also, the memory consumption will grow proportionally with the number of Data Tasks executed in parallel.

4.4.2 Leakage versus Performance Trade-off

The leakage factor k can be used as a tuning knob in all the strategies to trade-off leakage for performance. Table 2 presents the complexity of the main factors impacting performance when $k > 1$. We can see that increasing k benefits to all methods. In particular, for Adaptive, it reduces the number of enclaves and communications. For Reverse-and-replay, it reduces the number of communications. For Repartition-and-replay, it reduces all factors although in a smaller (logarithmic) proportion. Obviously, increasing k negatively impacts privacy since it increases the risk of object leakage. However, k does not impact the data set leakage.

	# of Data Tasks	# of communications	# of computations
Adaptive	$\frac{n}{k}$	$2 \cdot \frac{n}{k}$	n
Reverse-and-replay	2	$4 \cdot \frac{n}{k}$	$2 \cdot n$
Repartition-and-replay	$m \cdot \log_m(\frac{n}{k})$	$2 \cdot m \cdot \log_m(\frac{n}{k})$	$n \cdot \log_m(\frac{n}{k})$

Table 2: Complexity (depending on $n = |O_\sigma|$) of the main factors impacting the cost of the proposed strategies (for $1 < k < n$).

4.4.3 Limitations of the Proposed Solution

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 its 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 type of *cmp* does not allow parameters from the App (e.g., *cmp* is a similarity function 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.

Both Replay strategies rely on an assumption introduced in Section 4.3.1. Namely, if two deterministic Data Tasks DT_1^{cmp} and DT_2^{cmp} produce the same result for object o_j upon receiving the object sets O_1 and O_2, respectively, with $O_1 \cap O_2 = \{o_j\}$, then we assume that this result does not contain information about objects in $(O_1 \cup O_2) \setminus \{o_j\}$. In some cases this hypothesis does not hold and therefore, our replay strategies can no longer guarantee the leakage upper bound at object level (see Def. 2). The typical example is when there is redundancy in the data set. If two distinct objects of the database $o_x \in O_1$ and $o_y \in O_2$ have the same content ($o_x = o_y$), then both DT_1^{cmp} and DT_2^{cmp} are able to produce the same result for object o_j, but containing information about o_x and o_y, respectively. For such cases, adapted strategies have to be designed to enforce the same leakage upper bound at object level as a Decomposed execution with a leakage factor of $k = 1$. For instance, in the case of a redundant data set with a known level of redundancy, we could tune the value of m for the Repartition-and-Replay strategy such that there is at least one partition that does not contain redundant objects with a high probability.

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. And 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 make sense, but it is left as future work, as in this work, we wanted to be generic in terms of 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.

5 Validation

This section presents our extensive performance evaluation of the security building blocks and execution strategies introduced in Section 3 and Section 4, respectively. Our evaluation studies the efficiency and scalability of these proposals based on an implementation using Intel SGX, two real-world data sets and representative computations.

5.1 Experimental Setting

5.1.1 Experimental Software

We developed a Personal Data Management System prototype previously presented in [40]. It is programmed in C++ 17 using the Open Enclave [41] v0.16.1 Software Development Kit. This SDK aims at generalizing the development of enclave applications across several technologies of Trusted Execution Environments by providing a hardware-agnostic open source library. Open Enclave currently supports Intel SGX enclaves as well as a preview support for ARM TrustZone.

In its current state, our prototype is composed of approximately 12,000 lines of code and leverages SGX for both the Core and the Data Tasks. The Core includes an SQLite engine to store data. The secure channels between the Core and the Data Tasks are established using Mbed TLS [42].

The version of the prototype used to measure performances is accessible online[4].

5.1.2 Experimental Hardware

For all experiments, we used 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 v1.4.1. 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 entirely 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 overtime of approximately 10% due to a slower CPU, and are thus omitted. Finally, to capture

[4] https://gitlab.inria.fr/rcarpent/es-pdms-prototype

	Energy	GPS
Number of data points	2 075 259	24 876 978
Number of objects	34 587	18 670
Object size (data points)	60	1 332
Object size (bytes)	720	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 3: Considered Data and Query sets

the cases where the Core and the Data Tasks run on different machines, we simulated increased communication latency and slower bandwidth.

5.1.3 Data Sets

We use two public data sets of personal data (see Table 3). The first [43] (Energy) contains the electric power consumption of a household minute by minute over a period of four years. We split the data set into objects, each one being a time series containing the consumption of one hour (i.e., 60 data points). The second [44] data set (GPS) 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 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.

5.1.4 Query Sets

We give the App the right to request the execution of two UDFs, one for each data set. For the Energy data set, the App is able to receive the average of the energy consumption of the user for any time interval(s) provided by the App at execution time through σ. This corresponds to the *Energy* scenario described in Section 1. For the GPS data set, the 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 codes for Data tasks implementing a *cmp* function, one computing the integral of electric consumption time series and the other computing the length of GPS trajectories ; and two codes for *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. The same *cmp* and *agg* functions have also been integrated into the Core to enable a performance comparison of a basic architecture without Data Tasks as presented in Section 5.3.4.

5.1.5 Experimental Approach

Our experimental evaluation consists in three main steps. First, we measure in Section 5.2 the execution time of $agg \circ cmp$ using the Decomposed execution (presented in Section 3.3) with a maximum degree of decomposition (i.e., a leakage factor of $k = 1$) with varying selectivity (i.e., different σ). The objective is to understand the main bottlenecks of a straightforward application of the security building blocks. Second, we evaluate and compare in Section 5.3 the performance offered by the three proposed execution strategies, i.e., Adaptive, Reverse-and-Replay and Repartition-and-Replay (presented in Section 4) with varying selectivity on both data sets. We also study the impact of network latency on performance as well as the trade-off between security and performance by varying the leakage factor k. This enables us to observe which execution strategy performs best depending on the context (computation complexity, latency, security level, parallel execution, execution over a sequence of queries). Note that we have not implemented the sandbox, partly because we wanted to evaluate the performances of our strategies independently of it, and also because it necessitates a whole new study given the diversity of existing solutions [22, 23, 34–36]. Finally, Section 5.5 offers a performance overview for the proposed strategies varying the cost of the main factors impacting performance which cover a wide range of realistic scenarios. This enables us to select the best fitted strategy for a given context and infer the expected speedup compared with the other strategies.

In the rest of this section, execution time includes the different steps of the execution as follows:

1. the creation of the enclaves for all Data Tasks
2. the attestation of each Data Task by the Core
3. the transmission of the input objects to *cmp* Data Task(s), including encryption and decryption
4. the computation of *cmp* by the Data Task(s)
5. the transmission of the results to the Core, including encryption and decryption
6. the transmission of these results to one *agg* Data Task, including encryption and decryption
7. the computation of *agg* by this Data Task
8. the transmission of the final result to the Core, including encryption and decryption

The costs of communication between the App and the Core and of the SQL query processing by the Core are omitted since they are not relevant in the performance evaluation of the proposed strategies. Each created Data task enclave has a fixed size (i.e., 1MB of heap and 16KB of stack) to which we add a variable size corresponding to the total size of the objects that are processed by the Data task (i.e., 720 bytes per electric trace and 32KB on average per GPS trace). The extra cost caused by the paging mechanism of Intel SGX when the RAM is full (see [20]) is not considered in our measurements. That is, we inferred the time of executions requiring more than 94MB of RAM, i.e., the limit of RAM usable by enclaves on our CPU model, as this limit is meant to be largely extended and has already been set to 512GB per CPU socket in the last generation of Intel XEON Processors [45]. All reported execution times are the average of ten executions querying the same number of objects. In all

experiments, we consider a default value of the leakage factor $k = 1$ and a single execution thread (i.e., no parallel execution with multiple Data tasks) unless stated otherwise. For Repartition-and-Replay, we always set the value of m as computed from the Equation 1 to minimize the execution cost. Also, we consider a default setting wherein the Core and the Data tasks run on the same machine unless stated otherwise.

5.2 Performance of the Basic Decomposed Execution

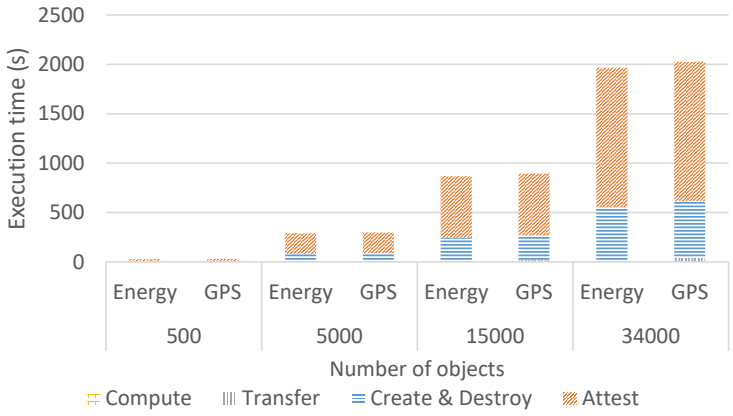
Figure 7a exposes the execution time of the Decomposed strategy (configured with the minimal leakage factor $k = 1$) with different input sizes from the Energy and GPS data sets. The cost increases linearly with the number of objects, while the execution time is similar between Energy and GPS. However, the execution time is very high even with a relatively low number of objects (e.g., it exceeds 1 sec. with only 17 Energy objects processed). Unsurprisingly, Decomposed execution is impractical when configured with the minimum leakage factor. From this figure we can already see that the cost is dominated by the enclave creation and attestation costs.

To have a better readability of the execution time breakdown, Figure 7b presents the same values but with a logarithmic scale. We observe that Data tasks' creation and attestation are the most time-consuming operations, representing more than 98% of the execution time. Indeed, the cost of creating Intel SGX enclaves is relatively high (at least 15ms in our tests) and increases with the amount of memory allocated for the enclave. Moreover, the Core has to establish a secure TLS channel with each Data task to authenticate it and to securely send the input data and retrieve the results. This attestation takes approximately 40ms per enclave in our tests. Both costs are multiplied by the number of Data tasks (one Data Task per object in this strategy configuration), hence the high overhead. In comparison, the cost induced by communications and computations of the user-defined functions (all executions of *agg* and *cmp* functions) are marginal. We also note at a more detailed level that the data computation related costs are 42 times higher on the GPS data set compared to Energy, because the GPS objects are larger. In conclusion, given the cost decomposition, minimizing the number of Data tasks involved in an execution is paramount.

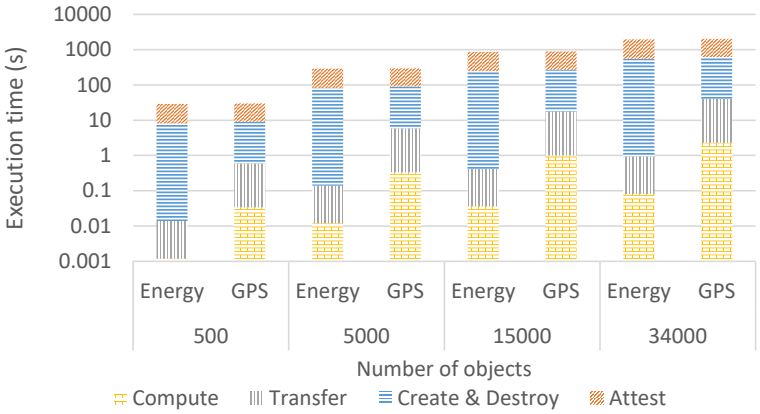
5.3 Performance of the Advanced Execution Strategies

Figure 8 compares the performance of Adaptive, Reverse-and-Replay and Repartition-and-Replay with a varying input size from the Energy and GPS data sets. The results show a major performance gain (note the logarithmic scale) of both replay strategies over Adaptive. Remember that Adaptive still needs to create a Data task for each input object when $k = 1$. On Energy, Repartition-and-Replay needs 7s to handle all 34,000 objects while Reverse-and-Replay spends 2s, thus being respectively 279 and 975 times faster than Adaptive taking 1966s. For 18,000 GPS objects, the speedup is 15 for Repartition-and-Replay and 25 for Reverse-and-Replay. Hence, by reducing the number of Data Tasks, the strategies leveraging replay offer a clear performance gain over Adaptive.

Regarding the two replay strategies, Reverse-and-Replay is significantly faster than Repartition-and-Replay on Energy while still offering some benefit on GPS. This shows



(a) Execution time breakdown



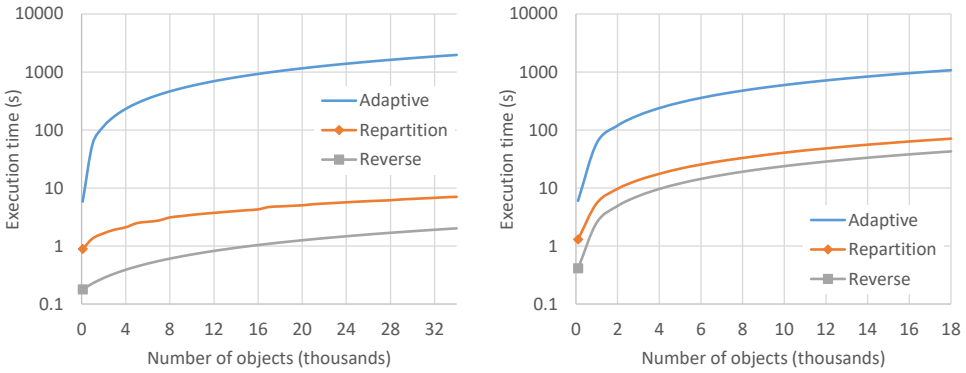
(b) Execution time breakdown (with logarithmic scale)

Fig. 7: Decomposed execution

that in this default configuration wherein all enclaves (Core and Data tasks) run on the same machine incurring thus minimal communication overhead, the main factor impacting performance remains the number of Data tasks. And Reverse-and-Replay has the best complexity from this viewpoint (see Table 1).

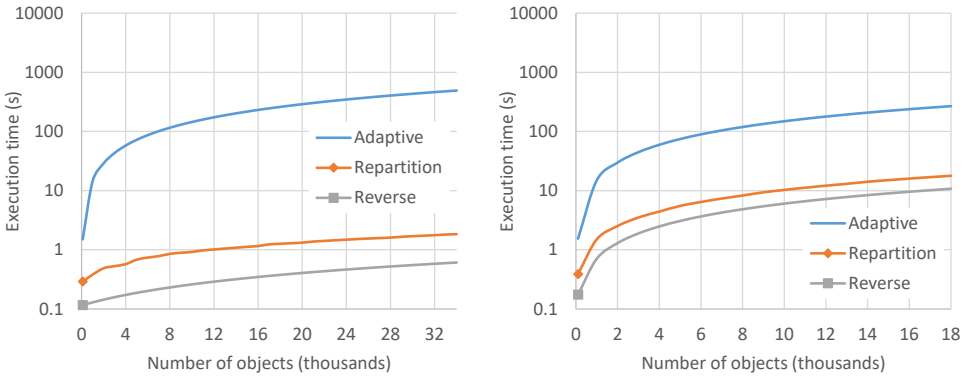
5.3.1 Parallel Execution of Data Tasks

To optimize performance, we can benefit from the parallelism capabilities of our CPU by executing several Data Tasks in parallel as discussed in Section 4.4.1. Out of the 6 available CPU cores, one is reserved for the Core. Besides, for a fair comparison, we use 4 CPU cores instead of the remaining 5 because the Reverse-and-Replay strategy can only be divided into an even number of Data Tasks: half of them computing the objects



(a) Energy (b) GPS

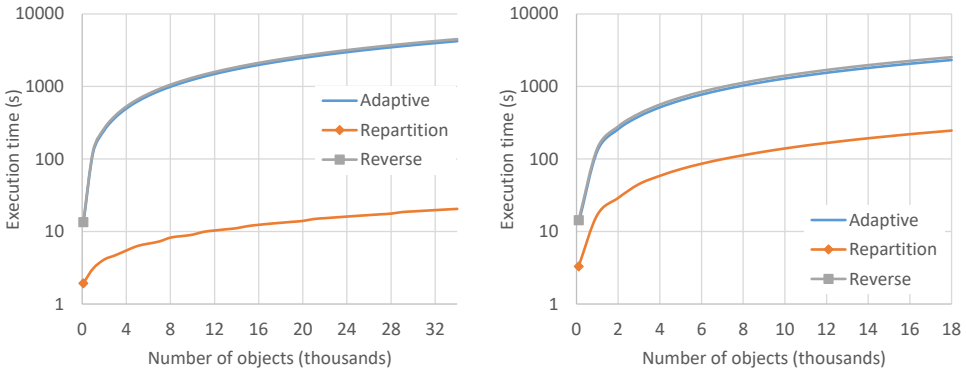
Fig. 8: Performance of the proposed strategies in the default setting



(a) Energy (b) GPS

Fig. 9: Performance of the proposed strategies with 4 concurrent Data Tasks

in one order, the other half in the reversed order. Figure 9 presents the execution time of the three strategies when four DT^{cmp} are executed in parallel. As expected, parallel execution benefits to all methods and, compared with a serial Data Task execution in Figure 8, it reduces proportionally the execution time, i.e., by a factor of 3.5 on average. At the same time, the gap between strategies remains proportionally the same. In particular, this degree of parallelism is not sufficient for Adaptive to reach reasonable performance, while it is still at least one order of magnitude slower than Reverse-and-Replay.



(a) Energy (b) GPS

Fig. 10: Performance of the proposed strategies with network latency (60Mbps bandwidth and 33ms latency)

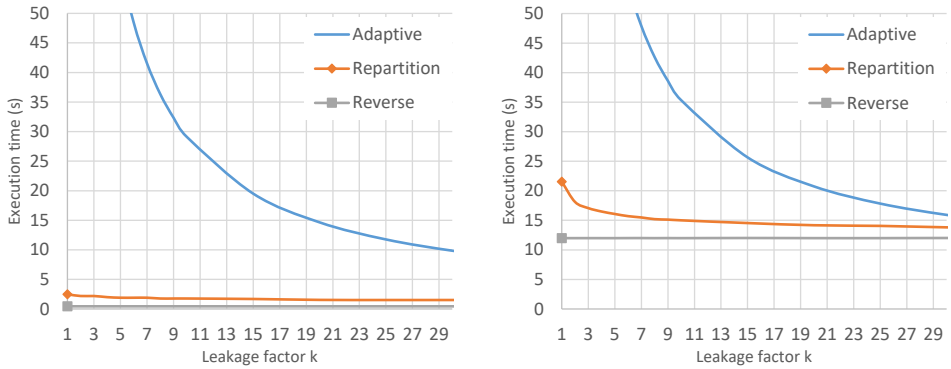
5.3.2 Impact of Network Latency

As discussed in Section 2.1.4, there are physical implementations of the PDMS architecture which leverage remote Data Tasks, hosted on Cloud servers, to carry out the computations. Figure 10 compares our execution strategies on both data sets in a distributed setting, e.g., wherein the Core runs on a PDMS user own device at home and the Data tasks are spawned as needed on a cloud server. We simulate a network link with a latency of 33ms and a bandwidth of 60Mbps representing the average values for domestic Internet in France [46]. In this setting, increased communication latency degrades the performance of all methods, but Reverse-and-Replay is the most affected of all having a performance similar to Adaptive. This is because of its pipelined data transmission between the Core and Data tasks, which adds latency for each computed object. With 34,000 Energy objects queried, Repartition-and-Replay is 599x more efficient than Reverse-and-Replay, spending 7s on the execution compared to 4489s. For this number of objects, Repartition-and-Replay is faster than Reverse-and-Replay when the latency is greater than 29μs. With 18,000 GPS objects queried, the speedup is 53x and Repartition-and-Replay is the fastest when the latency is greater than 250μs.

By efficiently sending the input objects in large batches, Repartition-and-Replay is the least impacted by latency and therefore is the preferred method in a distributed settings or generally whenever latency is involved.

5.3.3 Security vs. Performance Trade-off

The three proposed strategies can leverage the leakage factor k to increase computation efficiency. Obviously, this comes at the price of a higher risk of leakage at the object level as discussed in Section 4.4.2. Figure 11 shows the execution time with increasing k values on Energy and GPS data sets. Increasing k largely benefits



(a) Energy (b) GPS

Fig. 11: Performance of the proposed strategies with a varying leakage factor (5000 objects)

Adaptive and, in a much smaller proportion, Repartition-and-Replay, while it practically does not affect Reverse-and-Replay. The explanation is that k has a direct impact on the number of Data tasks for Adaptive and a smaller (logarithmic) impact on Repartition-and-Replay, while it only reduces the number of communications for Reverse-and-Replay (see Table 2). Nevertheless, despite of the increase in efficiency, Adaptive and Repartition-and-Replay are still behind Reverse-and-Replay, although we can observe a convergence or the execution times with large k values on GPS data.

5.3.4 Performance with Query Workloads

As expressed by our examples of scenarios in Section 1, the same App may need to compute many successive queries on different (and not necessarily disjoint) subsets of PDMS objects. Thus, it is important to study the performances of our strategies over a workload of queries corresponding to real-life scenarios.

Figure 12 shows the evolution of the execution time over 250 successive queries on the GPS data set. The workload was generated such that each query 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 $f = agg \circ cmp$ within the Core of the PDMS itself, thus without the costs of creating, attesting and destroying Data Tasks, nor the encryption and transmission of objects and results. This represents the case where the security of the UDF code has been fully verified, and thus can be added to the PDMS's TCB. While this may not be a realistic method for ensuring extensiveness nor compatible with the considered trust model, it establishes a lower bound in terms of performance and shows the overhead of using Data tasks.

From the first to the hundredth query, Reverse-and-Replay is 15x to 5x times more efficient than Adaptive, with Repartition-and-Replay being 2x more expensive then Reverse-and-Replay on average. The convergence of the execution times is due to the Result Reuse mechanism (see Section 4.2) which benefits all execution strategies, with

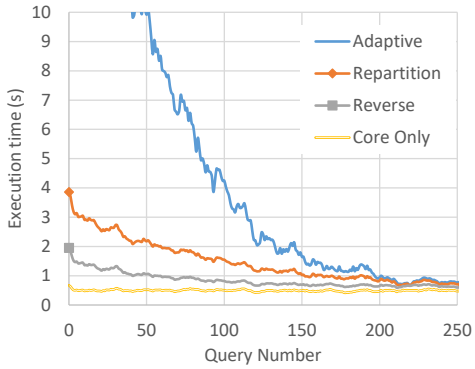


Fig. 12: Performance of the proposed strategies with successive queries – GPS

fewer objects to be computed as more queries are processed. The gap between both Replay strategies and Adaptive becomes insignificant (i.e., < 200ms) after 200 queries. On another hand, Reverse-and-Replay is at most 196% more expensive than 'Core only' for the first queries, representing the overhead of using Data Tasks, which ensure the extensiveness and security of the architecture.

On a query workload, both Replay strategies can be thus employed to ensure minimum leakage while being close to the ideal 'Core only' performance.

5.4 Improving the Cost of the Stateless Property

One of the major bottlenecks imposed by our security building blocks stems from the stateless property required by the Data tasks. The default mechanism to enforce this property on SGX requires to destroy, re-create and re-attest enclaves which is costly especially in the context of a decomposed execution implying several (if not many) Data tasks. Our Replay execution strategies are designed to improve the performance despite the high cost incurred by the stateless property by reducing the total number of Data tasks for a computation both in the classical single-machine context (i.e., Reverse-and-Replay) and in the distributed context (i.e., Repartition-and-Replay).

However, to further increase performance, more efficient implementations of stateless can be envisioned. For example, the authors of Ryoan [22] discuss a checkpoint-based mechanism where the initial state of a computation module inside an enclave is saved before receiving its input and restored after the computation, thus avoiding enclave re-creation. To do so, the sandbox "makes a full copy of the module's writable state and simply tracks that pages get modified. Only the contents of pages that were modified during input processing are restored" [22]. While their implementation is not available to grasp the exact underlying operations, they claim that this cost is "under 10ms" [22] in their use cases.

Such mechanisms are not free as they require extending the sandbox software and thus increasing the TCB. Yet, it is important to assess their potential performance impact on our strategies. To this end, we consider two more efficient mechanisms to

enforce the stateless property in Section 5.4.1 and infer their theoretical costs through simulations in Section 5.4.2.

5.4.1 Stateless Leveraging Memory-based Rollback

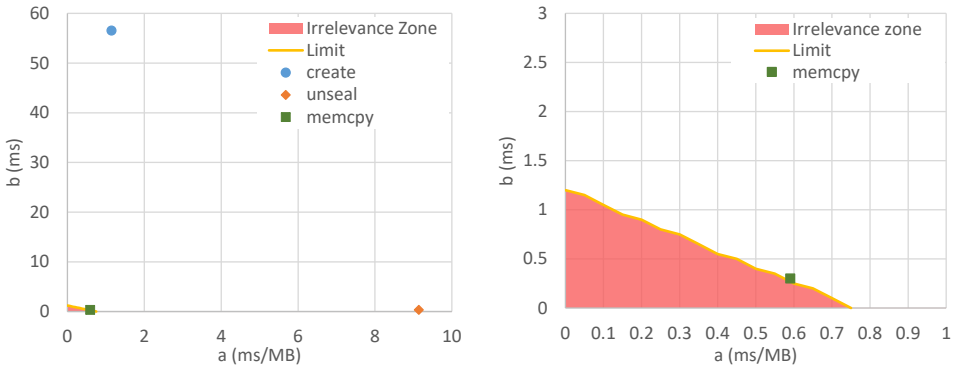
Inspired by the work in [22], we examine two mechanisms for implementing stateless based on rollback, i.e., creating an in-memory copy of the initial state of an enclave, which is then used whenever the initial state has to be restored.

Rollback with memcpy. The initial writable state of a Data Task (i.e. its heap, stack, .bss and .data memory segments) is backed-up by its sandbox after its launch using memcpy (or a similar primitive). The backup is stored in the enclave's memory and is isolated by the sandbox such that it cannot be accessed by the computing code itself. When needed, the sandbox overwrites the current writable state of the Data Task with the backup –again by using memcpy–, restoring its initial state and thus effectively enforcing the stateless property between computations. We estimate the minimal cost of this mechanism as the duration of the following operations: (i) the Core sends the request for rollback to the Data Task's sandbox; (ii) the sandbox performs the rollback at the speed of a memcpy then sends an acknowledgement to the Core. In our tests, communications last 150μs each and memcpy on SGX has a throughput of around 1700MB/s, as confirmed by [47]. In all likelihood, the true cost of rollback will be higher on a real implementation, with this estimation including only the key operations. Note that this mechanism requires each Data Task to be allocated with twice as much RAM as needed for the computation, to hold the backup of the writable state.

Rollback with unseal. To avoid allocating Data Tasks with a doubled size, we envisioned a second rollback mechanism, relying on the seal/unseal primitive of SGX [20]. Instead of storing the backup of the Data Task's writable state in the sandbox's memory (thus in SGX's trusted memory), we store it in non-enclave memory, leveraging SGX's security mechanism to encrypt data to be retrieved in the future. The backup of the writable state is sealed and placed in non-enclave memory. Upon rollback, the backup is unsealed and overwrites the current writable state of the Data Task. Unsealing has a throughput of 109MB/s in our tests.

Stateless as a linear cost. Other mechanisms to enforce the stateless property might be envisioned. To generalize, we abstract the cost of such mechanism as a linear function. We consider this cost to be linear because altering a state to not carry information depends on the size of that state and on a potentially fixed cost to carry out the mechanism. Let us define the stateless cost as $S(x) = a * x + b$ where x is the size of the state to be altered (in Megabytes), a the altering speed (in milliseconds/Megabytes) and b the fixed cost (in milliseconds). For example in the default mechanism with enclave re-creation, $b = 55$ms which corresponds to the creation (15ms) and attestation (40ms) of the smallest enclave in our tests. In addition, the enclave creation and destruction costs are proportional to the number of heap pages allocated to the enclave[5], with a factor of 1.14ms/MB in our tests.

[5] While it is also true for stack pages, our use cases only require variable heap, the stack of our enclaves being fixed to 16KB in all our experiments



(a) Overview

(b) Zoom on the irrelevance zone

Fig. 13: Relevance of the Replay strategies depending on the stateless cost

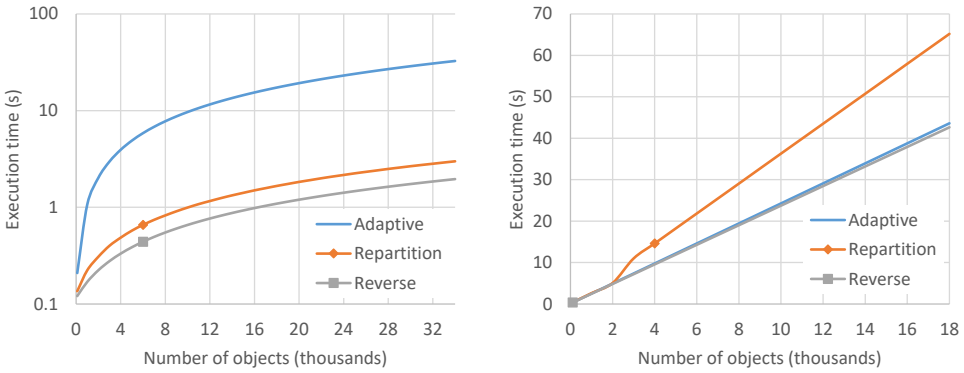
Figure 13 represents the relevance zone of our strategies with a linear stateless cost $S(x) = a * x + b$, a being on the x-axis and b on the y-axis. That is, for any stateless cost above the yellow limit (i.e., not situated within the red area, see Figure 13b), at least one Replay strategy has better performances than Adaptive. For a stateless cost within the red area, Adaptive has the most efficient performance. Also, the further the distance of the stateless cost from the irrelevance zone, the higher the performance benefit of the Replay strategies.

On this chart (see Figure 13a), *create* represents the default cost of the stateless mechanism with enclave re-creation, *memcpy* represents the cost of a rollback using memcpy and *unseal* of a rollback with unseal. We observe that event a stateless cost leveraging memcpy rollback, which represents a lower bound of the stateless cost, is outside de irrelevance zone, although close to the limit. This suggests that our Replay strategies can improve performance even in the case of a highly optimized stateless. Besides, the *create* and *unseal* stateless are located far from the irrelevance zone.

5.4.2 Impact of Rollback on Strategies

Figure 14 details the impact of the rollback mechanism on the strategies' performance in the extreme case, i.e., stateless has minimal cost by leveraging memcpy rollback, in the default setting (i.e., no network latency). For the Energy data set, Adaptive is naturally the most impacted with a 60x improvement compared to the default performance (i.e. Figure 8a) on 34,000 objects. On the another hand, the two Replay strategies do not rely on numerous enclaves and thus only gain a factor 2x for Repartition-and-Replay and almost nothing for Reverse-and-Replay. Nevertheless, Reverse-and-Replay is still 17 times faster than Adaptive.

For the GPS data set, Adaptive and Reverse-and-Replay have nearly the same performance (i.e., the cost of Adaptive is under 3% larger), both using 43s to compute 18,000 objects. Because GPS objects are larger and costlier to compute, the cost of



(a) Energy (b) GPS

Fig. 14: Strategies using rollback with memcpy

replay is higher than with Energy objects and gets closer to the cost of rollback. In this precise setting, strategies leveraging replay have no benefit over the baseline.

5.5 Performance Overview

This section concludes our detailed experimental evaluation with a performance overview of the proposed strategies correponding to wide range of scenarios and settings. The objective is to offer a synthetic view of the best fitted strategy depending on the context as well as the potential benefit it brings compared to Adaptive (which is our baseline strategy). For the context, we vary the main three factors that impact performance (as summarized in Table 1), i.e., the cost of having stateless enclaves, the communication latency and the complexity of the computation over objects.

For the stateless cost, we consider the three cases exposed in Section 5.4: (i) recreate a new enclave for each Data task (called 'Create'); (ii) rollback leveraging the unseal mechanism (called 'Unseal'); (iii) rollback using memcpy within the enclave memory (called 'Memcpy').

For the communication latency, we consider three different architectural settings (see Section 2.1.4): (i) a single machine running both the Core and the Data tasks (called 'Internal'); (ii) a distributed setting (called 'Local net') wherein the Core runs on a different machine from the Data tasks, and the machines are within a very low latency local network (e.g., a cloud infrastructure), i.e., we consider communications have a latency of 0.3ms and a throughput of 25Gbps; (iii) a distributed setting (called 'Internet') wherein the Core runs on a different machine from the Data tasks, and the machines are connected over the Internet, i.e., we consider communications have a latency of 33ms and a throughput of 60Mbps.

Finally, for the computation complexity we consider four computation classes from C1 to C4. C1 and C3 correspond to the average computation cost for the Energy data set and the GPS data set, respectively. C2 and C4 correspond to a computation cost which is 10x higher than for Energy and GPS, respectively. This way, we cover

Stateless	Comms.	Computation class			
		C1	C2	C3	C4
Create	Internal	Rev. (x652)	Rev. (x524)	Rev. (x24)	Rev. (x16)
	Local net.	Rep. (x188)	Rep. (x147)	Rep. (x67)	Rev. (x23)
	Internet	Rep. (x98)	Rep. (x94)	Rep. (x8)	Rep. (x7)
Unseal	Internal	Rev. (x199)	Rev. (x156)	Rev. (x8)	Rev. (x5)
	Local net.	Rep. (x110)	Rep. (x80)	Rep. (x23)	Rev. (x8)
	Internet	Rep. (x79)	Rep. (x76)	Rep. (x6)	Rep. (x5)
Memcpy	Internal	Rev. (x12)	Rev. (x10)	Rev. (x1)	Adaptive
	Local net.	Rep. (x37)	Rep. (x20)	Rep. (x6)	Rep. (x1)
	Internet	Rep. (x73)	Rep. (x70)	Rep. (x5)	Rep. (x4)

Table 4: Performance overview for 5000 objects (Rev. stands for Reverse-and-Replay and Rep. stands for Repartition-and-Replay)

computation with small to average costs (i.e., C1 and C2) to large and very large costs (i.e., C3 and C4).

Table 4 shows the performance overview of the strategies for computations over 5000 objects. Each table cell indicates the name of the best method (i.e., offering the lowest execution time) as well as the speedup with respect to Adaptive. A first observation is that Adaptive has the best performance in a single case: for very low latency stateless (memcpy) and communication (same machine) and very large objects. In this particular case, the execution cost is dominated by the object computation cost and Adaptive has the lowest complexity in this respect. Similarly, for two other close configurations, the advanced strategies offer similar execution times with Adaptive (i.e., the speedup is close to 1).

In the vast majority of all the other configurations, the advanced strategies are always faster offering much higher performance, i.e., a throughput which is often one or two order of magnitude larger than with Adaptive. Reverse-and-Replay is the best strategy whenever the communication latency is very low (i.e., Internal) regardless of the stateless or computation cost. With very large computations (i.e., C4), Reverse-and-Replay is also faster with moderate communication latecy (i.e., Local net) and larger stateless cost (i.e., Create or Unseal). Oppositely, Repartition-and-Replay is generally the fastest whenever communication latency is involved (i.e., Local net or Internet) regardless of the stateless cost and even the computation cost in most cases.

6 Related Work

Our solutions are designed to control the 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 [2, 3, 6, 8, 48], or locally on a personal device [2, 49, 50] –solutions such as [2] consider 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 [2, 49]. 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 [51, 52]. 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 [53, 54] rely on specific secure microcontroller –or trusted platform module (TPM)– running a lighweight data management engine connected to a mass-storage flash memory 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 [55].

Personal data management systems are therefore a promising tool, with some practical applications such as education [56], health [57, 58], the Internet-of-Things [59–61] and Artificial Intelligence [62]. However, we argue that the case for PDMS currently faces a dilemma. On one hand, there are solutions that fail to meet basic privacy requirements, thereby exposing data subjects to potential secondary uses by the cloud provider. On the other hand, there are proposals with limited functionalities that do not support advanced computations. Designing a solution that satisfies both privacy and extensibility requirements remains an open challenge. Our approach investigates a solution leveraging powerful secure hardware, such as Intel SGX, while enabling the Trusted Computing Base (TCB) to be coupled with advanced non-secure code modules [12]. This approach ensures privacy and provides control over potential data leakages.

DBMSs secured with TEEs. Many recent works [63–67] 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 [63] 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.

CryptSQLite [68] proposes to pair an SQLite engine with an SGX enclave to protect data confidentiality and integrity, while remote users undergo an initialisation phase where they exchange a symmetric key with the enclave. EnclaveDB [64] embeds

a –simplified– DBMS engine within an enclave and ensures data and query confidentiality, integrity and freshness. EdgelessDB [67] is an open-source project which can be deployed with docker and employs SGX to secure a MariaDB SQL engine. VeriDB [65] provides verifiability –correctness and completeness– of Select-Project-Join-Aggregate queries. IronSafe [69] presents a compelling approach leveraging both Intel SGX and ARM TrustZone TEEs to ensure confidentiality, integrity and freshness, while implementing Near Data Processing using low-power ARM CPUs inside storage devices paired with a more powerful Intel SGX-enabled host, resulting in efficient and secure SQL query processing. Finally, [66] introduces a Path ORAM protocol for SGX to avoid data leakage at query execution due to memory access pattern analysis.

While these proposals achieve various security properties like data confidentiality and potentially integrity/freshness with respect to the considered threat models, they all assume that the data processing code is trusted by the data owner. Support for User-Defined Functions/User-Defined Aggregate functions is not explicitly mentioned, but would imply the same trust assumption. Our work, on the contrary, makes an assumption of untrusted third-party function code for the data owner and with solutions to limit data leakage caused by this untrusted code.

Other proposals [22, 34] combine sandboxing and TEEs to secure data-oriented processing. For example, Ryoan [22] 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). A pioneer security measure was to isolate UDFs from the main DBMS process [70] by placing it inside a Java Virtual Machine to restrict its actions. Recent works such as [71] isolate the UDF code via virtualization (modified hypervisor), which can be applied to regular DBMSs but also to untrusted library usage or serverless contexts. DBMS products such as Snowflake [72, 73] or Google BigQuery [74] bring a solution to the case wherein a user u_1 who possesses data wants to authorize another user u_2 to execute a UDF on these data without having access to said data. These solutions notably disable some features of the query optimizer to avoid leaking data (e.g., no selection pushed down the execution tree to avoid revealing certain input data). In these cases, the UDF code as well as the server running it are trusted by the data owner u_1. This context however differs from ours since the UDF code is trusted and verified by the administrators with the ability to disable optimizer options.

Similarly to UDFs in DBMS, in the serverless computing model, because there is a delegation of computation, the confidentiality and integrity of the computation has to satisfy the clients' needs. In [75], the authors propose to leverage TEEs like Intel SGX to protect the data computed by those functions from an untrusted cloud provider, and rely on the Javascript Engine to isolate the UDFs running inside the same SGX enclave. Contrarily, [76] executes each function inside its own SGX enclave.

Information flow analysis. A body of literature [13, 15] exists in the context of information flow analysis to detect information leakage based on ensuring the property of *non-interference*. This 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 [77] considers the case of untrusted third-party machine learning applications running in a TEE. They introduce new definitions such as non-reversibility to capture other types of leakage. However, the proposal requires access to source code and the verification code must itself have to be part of the trusted code base, which is considered impractical in our context. It is also unclear how such proposal impacts computations other than machine learning algorithms.

Other works such as [14, 78] leverage code analysis, another information flow technique. Their solution is based on labelling the data from confidential sources and analysing their propagation through program variables, messages between applications, system library calls and file reads/writes. Similarly, the HomePad system [79] introduces a privacy-aware hub enforcing privacy policies encoded as prolog rules based on explicit information flow. Other solutions such as [51, 52] limit the leakage of Personally Identifiable Information (PII) such as phone number, IMEI or GPS coordinates, resorting to analysing the internet traffic. Although these solutions offer a way to control data leakage of potentially malicious code, they do not meet our requirements. Indeed, they cannot prevent leakage of PII that have been modified via ad-hoc encoding, compression or even encryption techniques.

7 Conclusion

In this paper, we addressed the important problem of securing data-oriented computations with untrusted code implementing User-Defined Functions. This is of particular interest in the Personal Data Management System (PDMS) context wherein applications requiring new, advanced functionalities are executed at the user-side. Our approach was to leverage an extensive and secure PDMS architecture using SGX enclaves as a basis for security. Then, we introduced new security building blocks which are necessary to limit the potential data leakage and showed the data leakage upper bound. Additionally, we proposed execution strategies allowing to drastically reduce the security cost compared with a straightforward execution strategy leveraging the building blocks. A prototype implementation [40] demonstrates the feasibility and benefits of the proposed solutions. Also, we showed that our strategies effectively improve execution cost in a wide rage of contexts (e.g., with different physical PDMS implementations involving low or high communication latency, or different computation complexity).

This contribution marks the first step towards a fully-featured Extensive and Secure PDMS. We hope this will open up new avenues for future research directions. Complementary security mechanisms can be employed, such as query restrictions

and additional trust assumptions, to further reduce data leakage. Furthermore, important research challenges include addressing complex selection predicates, supporting advanced calculations with parameters, and enabling distributed computation among multiple PDMS users while ensuring a balanced risk of data leakage between participants.

Acknowledgments

We thank Floris Thiant, former member of the PETRUS team, for his contribution to the prototype on which our experimentations are based.

Declarations

This work was supported by grant iPoP PEPR (ANR-22-PECY-0002).
The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] European Council: Regulation EU 2016/679 of the European Parliament and of the Council (2016). http://data.europa.eu/eli/reg/2016/679/2016-05-04 Accessed 2023-09-19

[2] Cozy: Cozy Cloud. https://cozy.io Accessed 2023-09-19

[3] Digi.me. https://digi.me Accessed 2022-09-27

[4] Sambra, A.V., Mansour, E., Hawke, S., Zereba, M., Greco, N., Ghanem, A., Zagidulin, D., Aboulnaga, A., Berners-Lee, T.: Solid: A Platform for Decentralized Social Applications Based on Linked Data. Technical report, MIT CSAIL & Qatar Computing Research Institute (2016)

[5] Solid. https://solidproject.org/ Accessed 2023-09-25

[6] BitsAbout.me. https://bitsabout.me Accessed 2023-09-25

[7] Prifina. https://www.prifina.com Accessed 2023-09-25

[8] mydex. https://mydex.org/ Accessed 2023-09-25

[9] Nextcloud. https://nextcloud.com Accessed 2023-09-25

[10] MyDataMood. https://mydatamood.com Accessed 2023-09-25

[11] MyData. https://mydata.org/ Accessed 2023-09-25

[12] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Pucheral, P., Sandu Popa, I., Scerri, G.: Personal data management systems: The security and functionality

standpoint. Information Systems **80**, 13–35 (2019) https://doi.org/10.1016/j.is.2018.09.002

[13] Sabelfeld, A., Myers, A.C.: Language-based information-flow security **21**(1), 5–19 (2003) https://doi.org/10.1109/JSAC.2002.806121

[14] Sun, M., Wei, T., Lui, J.C.S.: 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, pp. 331–342 (2016). https://doi.org/10.1145/2976749.2978343

[15] Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '99, pp. 228–241 (1999). https://doi.org/10.1145/292540.292561

[16] Dwork, C.: Differential Privacy. In: Automata, Languages and Programming, pp. 1–12 (2006). https://doi.org/10.1007/11787006_1

[17] Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: Security and privacy for mapreduce. In: Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, pp. 297–312 (2010)

[18] ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on Information Theory **31**(4), 469–472 (1985) https://doi.org/10.1109/TIT.1985.1057074

[19] Gentry, C.: A fully homomorphic encryption scheme. PhD thesis, Stanford university (2009)

[20] Costan, V., Devadas, S.: Intel SGX Explained. Cryptology ePrint Archive (2016). https://eprint.iacr.org/2016/086

[21] Pinto, S., Santos, N.: Demystifying arm TrustZone: A comprehensive survey. ACM Comput. Surv. **51**(6) (2019) https://doi.org/10.1145/3291047

[22] Hunt, T., Zhu, Z., Xu, Y., Peter, S., Witchel, E.: Ryoan: A distributed sandbox for untrusted computation on secret data. ACM Trans. Comput. Syst. **35**(4) (2018) https://doi.org/10.1145/3231594

[23] Weiser, S., Mayr, L., Schwarz, M., Gruss, D.: SGXJail: Defeating enclave malware via confinement. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 353–366 (2019)

[24] Carpentier, R., Sandu Popa, I., Anciaux, N.: 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, pp. 520–527

(2022). https://doi.org/10.5220/0011321900003269

[25] Carpentier, R., Sandu Popa, I., Anciaux, N.: Data Leakage Mitigation of User-Defined Functions on Secure Personal Data Management Systems. In: 34th International Conference on Scientific and Statistical Database Management. SSDBM 2022 (2022). https://doi.org/10.1145/3538712.3538741

[26] Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted Execution Environment: What It is, and What It is Not. In: 2015 IEEE Trustcom/BigDataSE/ISPA, pp. 57–64 (2015). https://doi.org/10.1109/Trustcom.2015.357

[27] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. Technical report, Intel Corporation (2013). https://perma.cc/B93F-JSTY

[28] ARM: Building a secure system using TrustZone technology. Technical report (2008). https://perma.cc/WE2E-WV2A

[29] Gueron, S.: Memory encryption for general-purpose processors. IEEE Security Privacy **14**(6), 54–62 (2016) https://doi.org/10.1109/MSP.2016.124

[30] Anati, I., Gueron, S., Johnson, S.P., Scarlata, V.R.: Innovative technology for CPU based attestation and sealing. Technical report, Intel Corporation (2013). https://perma.cc/CX5W-D56Z

[31] Brickell, E., Li, J.: Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. In: 2010 IEEE Second International Conference on Social Computing, pp. 768–775 (2010). https://doi.org/10.1504/IJIPSI.2011.043729

[32] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B.: Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 523–539 (2017)

[33] Schwarz, M., Weiser, S., Gruss, D.: Practical Enclave Malware with Intel SGX. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 177–196 (2019). https://doi.org/10.1007/978-3-030-22038-9_9

[34] Goltzsche, D., Nieke, M., Knauth, T., Kapitza, R.: AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting. In: Proceedings of the 20th International Middleware Conference. Middleware '19, pp. 123–135 (2019). https://doi.org/10.1145/3361525.3361541

[35] Park, J., Kang, S., Lee, S., Kim, T., Park, J., Kwon, Y., Huh, J.: Stockade: Hardware Hardening for Distributed Trusted Sandboxes. arXiv (2021). https:

//doi.org/10.48550/ARXIV.2108.13922

[36] Cui, J., Shinde, S., Sen, S., Saxena, P., Yuan, P.: Dynamic binary translation for SGX enclaves. ACM Trans. Priv. Secur. **25**(4), 1–40 (2022) https://doi.org/10.1145/3532862

[37] Carpentier, R., Sandu Popa, I., Anciaux, N.: Poster: Reducing Data Leakage on Personal Data Management Systems. In: 2021 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 716–718 (2021). https://doi.org/10.1109/EuroSP51992.2021.00057

[38] Brenner, S., Behlendorf, M., Kapitza, R.: Trusted execution, and the impact of security on performance. In: Proceedings of the 3rd Workshop on System Software for Trusted Execution. SysTEX '18, pp. 28–33 (2018). https://doi.org/10.1145/3268935.3268943

[39] Gjerdrum, A.T., Pettersen, R., Johansen, H.D., Johansen, D.: Performance of trusted computing in cloud infrastructures with intel SGX. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science, CLOSER, pp. 668–675 (2017). https://doi.org/10.1007/978-3-319-94959-8_1

[40] Carpentier, R., Thiant, F., Sandu Popa, I., Anciaux, N., Bouganim, L.: An Extensive and Secure Personal Data Management System Using SGX. In: Proceedings of the 25th International Conference on Extending Database Technology, EDBT (2022). https://doi.org/10.48786/edbt.2022.53

[41] Microsoft: Open Enclave SDK. https://openenclave.io Accessed 2023-09-25

[42] TrustedFirmware: Mbed TLS. https://tls.mbed.org/ Accessed 2023-09-25

[43] Georges Hebrail, Alice Berard: Individual household electric power consumption Data Set (2012). https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption Accessed 2023-09-25

[44] Microsoft Research Asia: GeoLife GPS Trajectories (2016). https://www.microsoft.com/en-us/download/details.aspx?id=52367 Accessed 2023-09-25

[45] Intel: 3rd Gen Intel Xeon Scalable Processors Brief (2021). https://www.intel.com/content/www/us/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html Accessed 2023-09-25

[46] nPerf: Baromètre des connexions internet fixes en france métropolitaine. Technical report (2020). https://perma.cc/DP8V-5ABT

[47] Hasan, A., Riley, R., Ponomarev, D.: Port or Shim? Stress Testing Application Performance on Intel SGX. In: 2020 IEEE International Symposium on Workload Characterization (IISWC), pp. 123–133 (2020). https://doi.org/10.1109/IISWC50251.2020.00021

[48] Personium. https://personium.io Accessed 2023-09-25

[49] Montjoye, Y.-A., Shmueli, E., Wang, S.S., Pentland, A.S.: openPDS: Protecting the Privacy of Metadata through SafeAnswers. PLOS ONE **9**(7), 1–9 (2014) https://doi.org/10.1371/journal.pone.0098790

[50] Chaudhry, A., Crowcroft, J., Howard, H., Madhavapeddy, A., Mortier, R., Haddadi, H., McAuley, D.: Personal Data: Thinking Inside the Box. Aarhus Series on Human Centered Computing **1**(1) (2015) https://doi.org/10.7146/aahcc.v1i1.21312

[51] Novak, E., Aung, P.T., Do, T.: VPN+ Towards Detection and Remediation of Information Leakage on Smartphones. In: 2020 21st IEEE International Conference on Mobile Data Management (MDM), pp. 39–48 (2020). https://doi.org/10.1109/MDM48529.2020.00025

[52] Ren, J., Rao, A., Lindorfer, M., Legout, A., Choffnes, D.: 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, pp. 361–374 (2016). https://doi.org/10.1145/2906388.2906392

[53] Allard, T., Anciaux, N., Bouganim, L., Guo, Y., Le Folgoc, L., Nguyen, B., Pucheral, P., Ray, I., Ray, I., Yin, S.: Secure personal data servers: a vision paper. Proceedings of the VLDB Endowment **3**(1-2), 25–35 (2010) https://doi.org/10.14778/1920841.1920850

[54] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Sandu Popa, I., Pucheral, P.: Trusted cells: A sea change for personal data services. In: Proceedings of the 6th Biennal Conference on Innovative Database Research (CIDR) (2013)

[55] Anciaux, N., Bouganim, L., Pucheral, P.: Future trends in secure chip data management. Bulletin of the Technical Committee on Data Engineering **30**(3), 49–57 (2007)

[56] Mitra, S., Gupta, S.: Mobile learning under personal cloud with a virtualization framework for outcome based education. Education and Information Technologies **25**(3), 2129–2156 (2020) https://doi.org/10.1007/s10639-019-10043-z

[57] Anciaux, N., Benzine, M., Bouganim, L., Jacquemin, K., Pucheral, P., Yin, S.: Restoring the Patient Control over Her Medical History. In: 2008 21st IEEE International Symposium on Computer-Based Medical Systems, pp. 132–137 (2008). https://doi.org/10.1109/CBMS.2008.101

[58] Akter, M., Gani, A., Rahman, M.O., Hassan, M.M., Almogren, A., Ahmad, S.: Performance Analysis of Personal Cloud Storage Services for Mobile Multimedia Health Record Management. IEEE Access **6** (2018) https://doi.org/10.1109/ACCESS.2018.2869848

[59] Lallali, S., Anciaux, N., Popa, I.S., Pucheral, P.: Supporting secure keyword search in the personal cloud. Information Systems **72** (2017) https://doi.org/10.1016/j.is.2017.09.003

[60] Anciaux, N., Lallali, S., Sandu Popa, I., Pucheral, P.: A Scalable Search Engine for Mass Storage Smart Objects. Proc. VLDB Endow. **8**(9), 910–921 (2015) https://doi.org/10.14778/2777598.2777600

[61] Perera, C., Wakenshaw, S., Baarslag, T., Haddadi, H., Bandara, A., Mortier, R., Crabtree, A., Ng, I., McAuley, D., Crowcroft, J.: Valorising the IoT Databox: Creating Value for Everyone (2016) https://doi.org/10.48550/ARXIV.1609.03312

[62] Meurisch, C., Bayrak, B., Mühlhäuser, M.: Privacy-Preserving AI Services Through Data Decentralization. In: Proceedings of The Web Conference 2020. WWW '20, pp. 190–200 (2020). https://doi.org/10.1145/3366423.3380106

[63] Microsoft: Azure SQL - Always encrypted with secure enclaves (2019). https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-enclaves Accessed 2021-11-04

[64] Priebe, C., Vaswani, K., Costa, M.: EnclaveDB: A secure database using SGX. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 264–278 (2018). https://doi.org/10.1109/SP.2018.00025

[65] Zhou, W., Cai, Y., Peng, Y., Wang, S., Ma, K., Li, F.: VeriDB: An SGX-Based Verifiable Database. In: Proceedings of the 2021 International Conference on Management Of Data. SIGMOD/PODS '21, pp. 2182–2194 (2021). https://doi.org/10.1145/3448016.3457308

[66] Han, Z., Hu, H.: ProDB: A memory-secure database using hardware enclave and practical oblivious RAM. Information Systems **96** (2021) https://doi.org/10.1016/j.is.2020.101681

[67] Edgeless Systems: EdgelessDB (2021). https://www.edgeless.systems/products/edgelessdb/ Accessed 2021-11-02

[68] Wang, Y., Shen, Y., Su, C., Ma, J., Liu, L., Dong, X.: Cryptsqlite: Sqlite with high data security. IEEE Transactions on Computers **69**(5), 666–678 (2019) https://doi.org/10.1109/TC.2019.2963303

[69] Unnibhavi, H., Cerdeira, D., Barbalace, A., Santos, N., Bhatotia, P.: Secure and Policy-Compliant Query Processing on Heterogeneous Computational Storage Architectures. In: ACM SIGMOD/PODS International Conference on Management of Data 2022 (2022). https://doi.org/10.1145/3514221.3517913

[70] Godfrey, M., Mayr, T., Seshadri, P., Eicken, T.: Secure and portable database extensibility. In: Proceedings of the 1998 ACM SIGMOD International Conference

on Management of Data. SIGMOD '98, pp. 390–401. https://doi.org/10.1145/276304.276339

[71] Wanninger, N.C., Bowden, J.J., Shetty, K., Garg, A., Hale, K.C.: Isolating functions at the hardware limit with virtines. In: Proceedings of the Seventeenth European Conference on Computer Systems. EuroSys '22, pp. 644–662. https://doi.org/10.1145/3492321.3519553

[72] Snowflake: Secure UDF (2019). https://docs.snowflake.com/en/sql-reference/udf-secure.html Accessed 2021-11-04

[73] Herzberg, B., Cohen, Y.: Secure data sharing with snowflake. In: Snowflake Security, pp. 163–175 (2022). https://doi.org/10.1007/978-1-4842-7389-0_8

[74] Google: Google BigQuery - Authorized Functions (2011). https://cloud.google.com/bigquery/docs/authorized-functions Accessed 2021-11-04

[75] Brenner, S., Kapitza, R.: Trust more, serverless. In: Proceedings of the 12th ACM International Conference on Systems and Storage. SYSTOR '19, pp. 33–43 (2019). https://doi.org/10.1145/3319647.3325825

[76] Trach, B., Oleksenko, O., Gregor, F., Bhatotia, P., Fetzer, C.: Clemmys: Towards secure remote execution in FaaS. In: Proceedings of the 12th ACM International Conference on Systems and Storage. SYSTOR '19, pp. 44–54 (2019). https://doi.org/10.1145/3319647.3325835

[77] Zhang, R., Zhang, N., Moini, A., Lou, W., Hou, Y.T.: PrivacyScope: Automatic analysis of private data leakage in TEE-protected applications. In: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pp. 34–44 (2020). https://doi.org/10.1109/ICDCS47774.2020.00013

[78] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. **32**(2) (2014) https://doi.org/10.1145/2619091

[79] Zavalyshyn, I., Duarte, N.O., Santos, N.: HomePad: A Privacy-Aware Smart Hub for Home Environments. In: 2018 IEEE/ACM Symposium on Edge Computing (SEC), pp. 58–73 (2018). https://doi.org/10.1109/SEC.2018.00012