Local Personal Data Processing with Third Party Code and Bounded Leakage

Robin Carpentier, Iulian Sandu Popa, Nicolas Anciaux

Local Personal Data Processing with Third Party Code and Bounded Leakage

Robin Carpentier[1,2], Iulian Sandu Popa[1,2] and Nicolas Anciaux[2,1]
[1] *University of Versailles-Saint-Quentin-en-Yvelines, Versailles, France*
[2] *Inria Saclay-Île-de-France, Palaiseau, France*

Keywords:
Personal Data Management Systems, User Defined Functions, Bounded Leakage.

Abstract:
Personal Data Management Systems (PDMSs) provide individuals with appropriate tools to collect, manage and share their personal data under control. A founding principle of PDMSs is to move the computation code to the user's data, not the other way around. This opens up new uses for personal data, wherein the entire personal database of the individuals is operated within their local environment and never exposed outside, but only aggregated computed results are externalized. Yet, whenever arbitrary aggregation function code, provided by a third-party service or application, is evaluated on large datasets, as envisioned for typical PDMS use-cases, can the potential leakage of the user's personal information, through the legitimate results of that function, be bounded and kept small? This paper aims at providing a positive answer to this question, which is essential to demonstrate the rationale of the PDMS paradigm. We resort to an architecture for PDMSs based on Trusted Execution Environments to evaluate any classical user-defined aggregate PDMS function. We show that an upper bound on leakage exists and we sketch remaining research issues.

1 Introduction

Personal Data Management Systems (PDMSs) are emerging, 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, Personium.io, Solid/PODS [Sambra et al., 2016] to name a few, and to large initiatives such as Mydata.org supported by data protection agencies. Such initiatives have been made possible by the successive steps taken in recent years to give individuals new ways to retrieve and use their personal data. In particular, the right to data portability in the European Union allows individuals to retrieve their personal data from different sources (e.g., health, energy, GPS traces, banks).

Context. Traditional solutions to handle computations involving user's data usually requires them to send those data to a third party application or service that performs the computation. In contrast, a PDMS can receive a third-party computation code and evaluate it locally. This introduces a new paradigm where only aggregated computed results are externalized, safeguarding user's control on their data. The example presented below illustrates this new paradigm.

> ***'Green bonus' example.*** Companies want to reward their employees having an ecological conduct, and thus monthly compute a green bonus (financial incentive) based on the number of commutes by bike. To this end, an employee collects her GPS traces within her PDMS from a reliable service (e.g., Google Maps). The traces are then processed locally and the result is delivered to the employer with compliance proof.

In this example, the PDMS must support ad hoc code specified by the employer while preventing the disclosure of personal data. Thus, the PDMS must offer both *extensiveness* and *security*. Several similar scenarios can be envisioned for service offers based on statistical analysis of historical personal *factual* data related to a user, e.g., health services (medical and prescription history), energy services (electric traces), car insurance (GPS traces), financial services (bank records).

Objective. This position paper aims at

analysing the conflict between extensiveness and security on PDMSs. Supporting a code created by a third party run by the PDMS to process large volumes of personal data (typically an aggregation) calls for extensiveness. However, the *PDMS user* (owner of the data and of the PDMS) does not trust this code and the PDMS must ensure security of the data during the processing. More precisely, we search to control potential information leakage in legitimate query results during successive executions of such code.

Limitations of existing approaches. Traditional DBMSs ensure extensiveness of computations by the support of user-defined functions (UDFs). But UDF security relies on administrators, e.g., checking/auditing UDF code and their semantics. In contrast, PDMSs are administered by laymen users. In addition, the function code is implemented by third-parties, requires authorized access to large data volumes and externalizes results. Therefore, the use of traditional UDF solutions is proscribed. Another approach would be to use anonymization or differential privacy [Dwork, 2006] to protect the input of the computed function, but this method is not generic thus undermining the extensiveness property and can hardly preserve result accuracy. Similarly, employing secure multiparty cryptographic computation techniques can hurt genericity (e.g., semi-homorphic encryption [ElGamal, 1985]) or performance (e.g., fully homomorphic encryption [Gentry, 2009]), and cannot solve the problem of data leakage through legitimate results computed by untrusted code.

Proposed approach. We consider split execution techniques 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 orchestrating the evaluation to mitigate personal data leakage and ensure security. In our architecture, the security relies on hardware properties provided by Trusted Execution Environments (TEEs), such as Intel SGX [Costan and Devadas, 2016] or ARM TrustZone [Pinto and Santos, 2019], and sandboxing techniques within enclaves, like Ryoan [Hunt et al., 2018] or SGXJail [Weiser et al., 2019].

Contributions. Our contribution is twofold. First, we formalize the threat and computation models adopted in the PDMS context. Then, we introduce three security building blocks to bound the information leakage from user-defined computation results on large personal datasets.

2 Background

2.1 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 potentially untrusted code. We proposed in [Anciaux et al., 2019] a three-layers logical 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 Fig. 1).

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, and hence provides the basic DBMS operations required to ensure data confidentiality, integrity, and resiliency. It therefore implements a data storage module, a policy enforcement module to control access to PDMS data and a basic query module (as needed to evaluate simple selection predicates on database objects metadata) and a communication manager for securing data exchanges between the architecture layers and with the Apps.

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 the applications (large code base) and their execution environment (e.g., web browser) make them vulnerable. Therefore, no security assumption is made: Apps manipulate only authorized data resulting from Data tasks but have no privilege on raw data.

2.2 Security Properties

The specificity of our architecture is to remove from local or remote Apps any sensitive data-oriented computation, delegating it to Data tasks running UDFs under the control of the Core. App leverages an *App manifest* which includes essen-

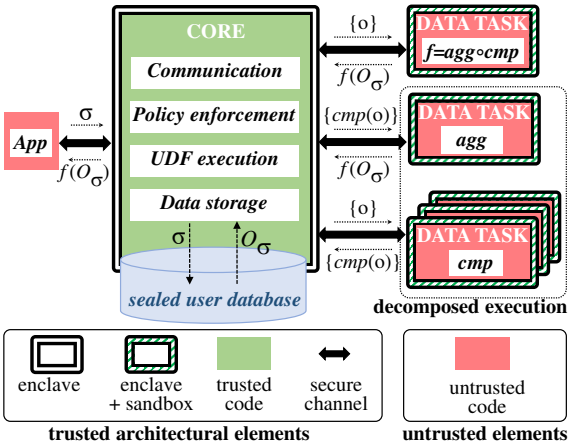

Figure 1: Computation and Threat Models.

tial information about the UDFs to be executed by the App, including their purpose, authorized PDMS objects and size of results to be transmitted to the App. The manifest should be validated, e.g., by a regulatory agency or the App marketplace, and approved by the PDMS user at install time before the App can call corresponding functions. Specifically, our system implements the following architectural security properties:

P1. Enclaved Core/Data tasks. The Core and each Data task run in *individual enclaves* protecting the confidentiality of the data manipulated and the integrity of the code execution from the untrusted execution environment (application stack, operating system). Such properties are provided by TEEs, e.g., Intel SGX, which guarantees that (i) enclave code cannot be influenced by another process; (ii) enclave data is hidden from any other process (RAM encryption); (iii) enclave can prove that its results were produced by its genuine code [Costan and Devadas, 2016]. Besides, the code of each Data task is sandboxed [Hunt et al., 2018] within its enclave to preclude any voluntary data leakage outside the enclave by a malicious Data task function code. Such Data task *containment* can be obtained using existing software such as Ryoan [Hunt et al., 2018] or SGXJail [Weiser et al., 2019] which provide means to restrict enclave operations (bounded address space and filtered syscalls).

P2. Secure communications. All the communications between Core, Data tasks and Apps are classically secured using TLS to ensure authenticity, integrity and confidentiality. Because the inter-enclave communication channels are secure and attested, the Core can guarantee to 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 'Green bonus' example, a Data task running a UDF computing the number of bike commutes during a certain time interval will only be supplied by the Core with the necessary GPS traces (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., number of bike commutes) 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.

3 Related Works

Personal Data Management Systems. PDMSs (also called Personal Clouds or PIMS) are hardware and/or software platforms enabling users to retrieve and exploit their personal data (e.g., banking, health, energy consumption, IoT). The user is considered the sole administrator of their PDMS, whether it is hosted remotely within a personal cloud (Cozy Cloud, BitsaboutMe, Personium, Mydex), or locally on a personal device (Cozy Cloud, [de Montjoye et al., 2014, Chaudhry et al., 2015]).

Existing solutions include support for advanced data processing (e.g., statistical processing or machine learning on time series or images) by means of applications installed on the user's PDMS platform (Cozy Cloud, [de Montjoye et al., 2014]). Data security and privacy are essentially based on code openness and community audit (expert PDMS users) to minimize the risk of data leakage. Automatic network control mechanisms may also help identifying suspicious data transfers [Novak et al., 2020]. However, no guarantee exists regarding the amount of PDMS data that may be leaked to external parties through seemingly legitimate results.

DBMSs secured with TEEs. Many recent works [Priebe et al., 2018, Zhou et al., 2021, Han and Hu, 2021], EdgelessDB or Azure SQL adapt existing DBMSs to the constraints of TEEs, to enclose the DBMS engine and thus ben-

efit from TEE security properties. For example, EnclaveDB [Priebe et al., 2018] and EdgelessDB embed a –minimalist– DBMS engine within an enclave and ensure data and query confidentiality, integrity and freshness, while VeriDB [Zhou et al., 2021] provides verifiability of database queries.

These proposals consider the DBMS code running in the enclaves to be trusted by the DBMS owner –or administrator–. Support for untrusted UDF/UDAF-like functions is not explicitly mentioned, but would imply the same trust assumption for the UDF code. Our work, on the contrary, makes an assumption of untrusted third-party UDF code for the database owner.

Other proposals [Hunt et al., 2018, Goltzsche et al., 2019] combine sandboxing and TEEs to secure data-oriented processing. For example, Ryoan [Hunt et al., 2018] protects, on the one hand, the confidentiality of 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 DBMSs. DBMSs support the evaluation of third-party code via UDFs. Products such as Snowflake or Google BigQuery consider the case of *secure* or *authorized* UDFs, where users with UDF execution privilege do not have access rights to the input data. This context is different from ours because the UDF code is trusted and verified by the administrators.

Information flow analysis. Several works [Sabelfeld and Myers, 2003, Sun et al., 2016, Myers, 1999] deal with information flow analysis to detect information leakage, but these are essentially based on insuring the *non-interference* property, which guarantees that if the inputs to a function f are sensitive, no public output of that function can depend on those sensitive inputs. This is not applicable in our context, since it is legitimate –and an intrinsic goal of running f– for App 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.

4 Computing and Threat Models

4.1 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 above). The model should not impact the application usages, while allowing to address the legitimate privacy concerns of the PDMS users. Hence, we exclude from the outset UDFs which are permitted by construction to extract large sets of raw database objects (like SQL select-project-join queries). Instead, we consider UDFs (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 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 function cmp computes each object o of the input. For instance, cmp can compute the integral of a time series indicating the electricity consumption or the length of a GPS trace stored in o, while agg can be a typical aggregate function applied subsequently on the set of cmp results. Besides, we consider that the result of cmp over any object o has a fixed size in bits of $||cmp||$ with $||cmp|| << ||o||$ (e.g., in the examples above about time series and GPS traces, cmp returns a single value –of small size– computed from the data series –of much larger size– stored in o). For simplicity and without lack of generality, we focus in the rest of the paper on a single App a and computation function f.

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 at App install time. 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. Function f

computes $agg(\{cmp(o)\}_{o \in O_\sigma})$, 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.

4.2 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 produced 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 $O_\sigma \in O$, 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 2.2) 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 used to enforce P1 to P3. Fig. 1 illustrates the computation model considered for the UDFs and the trust assumptions on each of the architectural elements of the PDMS involved in the evaluation.

Note that to foster usage, we impose no restrictions on the σ predicate and on the App query budget (see Section 6). In addition, we do not consider any semantic analysis or auditing of the code of f since this is not realistic in our context (the layman PDMS user cannot handle such tasks) and also mostly complementary to our work as discussed in Section 3.

4.3 Problem Formulation

Our objective is to address the following question: 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?

Answering this question is critical to bolster the PDMS paradigm. A positive conclusion is necessary to justify a founding principle for the PDMS, insofar as bringing the computational function to the data, would indeed provide a quantifiable privacy benefit to PDMS users.

Let us consider a simple attack example:

> ***Attack example.*** The code for f, instead of the expected purpose which users consent to (e.g., computing bike commutes for the 'Green Bonus'), 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 a same set O of database objects as input, the attacker obtains after each execution a new chunk of information about O encoded on $||f||$ bits. The attacker could thus reconstruct O by assembling the received $leak$ after at most $n = \frac{||O||}{||f||}$ successive executions, with $||O||$ the size in bits needed to encode the information of O.

To address the formulated problem, we introduce metrics inspired by traditional information flow methods (see e.g., [Sabelfeld and Myers, 2003]). We denote by $||x||$ the amount of information in x, measured by the number of bits needed to encode it. For simplicity, we consider this value as the size in bits of the result if x is a function and as its footprint in bits if x is a database object or set of objects. We define the leakage $L_f(O)$ resulting from successive executions of a function f on objects in O allowed to f, as follows:

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

5 Countermeasure Building Blocks

We introduce security building blocks to control data leakage in the case of multiple executions of a UDF $f = agg \circ cmp$ and show that bounded leakage can be guaranteed.

5.1 Definitions

Since attackers control the code of f, an important lever that they can exploit is data persistency. For instance, in the Attack example (Section 4.3), f maintains a variable *leak* to avoid leaking the same data twice. A first building block is hence to rely on *stateless* Data Tasks.

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

In the particular case of executions dealing with identical inputs, an attacker could exploit randomness provided by its environment to leak different data. To further reduce leakage, we enforce a second building block: 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 increasing Data set leakage in the case of identical executions (enforcing Def. 1).

Regardless if Data Tasks are stateless and deterministic, attackers can leak new data with each execution of f by changing the selection predicate σ. Attackers can also focus their leakage on a specific object o, by executing f on different input sets each containing o. To mitigate such attacks, we introduce a third building block by decomposing the execution of $f = agg \circ cmp$ into a set of Data Tasks: one Data Task DT^{agg} executes the code of agg and a set of Data Tasks $\{DT_i^{cmp}\}_{i>0}$ executes cmp on a partition of the set of authorised objects O, each part P_i being of maximum cardinality k. The partitioning of O has to be static (independent of the input O_σ of f), so that any object is always processed in its same partition P_i across executions.

Definition 5. Decomposed Data Tasks. Let $P(O) = \{P_i\}$ be a *static* partition of the set of objects O, authorized to function $f = agg \circ cmp$, such that any part P_i is of maximum cardinality $k > 0$ (k being fixed at install time). A decomposed Data Tasks execution of f over $O_\sigma \subseteq O$ involves a set of Data Tasks $\{DT_i^{cmp}, P_i \cap O_\sigma \neq \emptyset\}$, with each DT_i^{cmp} executing cmp on a given part 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 $res_i^{cmp} = \{cmp(o), o \in P_i\}$ to the Core. The Core discards all results corresponding to objects $o \notin O_\sigma$, i.e., not part of the computation. One last Data Task DT^{agg} is used to aggregate the union of the results of the computation, i.e., $\cup_i \{res_i^{cmp} = \{cmp(o), o \in P_i \cap O_\sigma\}\}$ and produce the final result.

5.2 Enforcement

On SGX, statelessness (Def 3) 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).

To enforce determinism (Def 4), Data Task containment (security property *P1*) can be leveraged 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", i.e., the random numbers 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 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.

Finally, to enforce decomposition (Def 5), it is sufficient to add code to the Core that implements an execution strategy consistent with this definition.

5.3 Leakage Analysis

Stateless Only. A corrupted computation code running as a Stateless Data Task may leverage randomness to maximize the leakage rate. In the Attack example, at each execution, f_{leak} would select a random fragment of O to produce a *leak* –even if the same query is run twice on the same input–. Considering a uniform leakage function, the probability of producing a new *leakage* decreases with the remaining amount of data –not yet disclosed– present in the data task input O.

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

With Decomposition. Any computation involves one or several partitions P_i of O. Information about objects in any P_i can only leak into up to the k results produced by DT_i^{cmp}. The parameter k is called *leakage factor*. Leveraging stateless deterministic Data Tasks, the result set $\{cmp(o)\}_{o \in P_i}$ is guaranteed to be unique for any o. Hence, the data set leakage for each partition P_i is bounded by $||cmp|| \cdot |P_i|$, $\forall P_i$ regardless of the number of the computations involving $o \in P_i$. Consequently, the Data set leakage over large numbers n of computations is also bounded: ${L_f}^{n \to +\infty}(O) \leq \sum_i ||cmp|| \cdot |P_i| = ||cmp|| \cdot |O|$. Regarding Object leakage, for any P_i, the attacker has the liberty to choose the distribution of the $|P_i|$ leak fragments among the objects in P_i. At the extreme, all $|P_i|$ fragments can concern a single object in P_i. For any object $o \in P_i$, the Object leakage is thus bounded by ${L_f}^{n \to +\infty}(o) \leq min(||cmp|| \cdot k, ||o||)$.

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

'Green bonus' leakage analysis. Let us assume that $||cmp|| = 1$ (i.e., 1 bit to indicate if a GPS trace is a bike commute or not), $||agg|| = 6$ (i.e., 6 bits to count the number of monthly bike commutes with a maximum admitted value of 60) and $||o|| = 600$ (i.e., each GPS trace is encoded with 600 bits of information). Without any countermeasure on f, an attacker needs 100 queries to obtain an object o. With a stateless 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 deterministic 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 containing o. Finally, with a fully decomposed execution of f (i.e., $k = 1$), only $||cmp|| = 1$ bit of o can be leaked regardless of the number of queries.

6 Research Challenges

The introduced building blocks allow an evaluation of f with low and bounded leakage. However, their straightforward implementation may lead to prohibitive execution cost with large data sets, mainly because (i) too many Data Tasks must be allocated at execution (up to one per object $o \in O_\sigma$ to reach the minimal leakage) and (ii) many unnecessary computations are needed (objects $o' \notin O_\sigma$ must be processed given the 'static' partitioning if they belong to any part containing an object $o \in O_\sigma$). Hence, a first major research challenge is to devise evaluation strategies having reasonable cost, while achieving low data leakage.

The security guarantees of our strategies are based on the hypothesis that the Core is able to evaluate the σ selection predicates of the App. This is a reasonable assumption if basic predicates are considered over some objects metadata (e.g., temporal, object type or size, tags). However, because of the Core minimality, it is not reasonable to assume the support for more complex selections within the Core (e.g., spatial search, content-based image search). 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 issue is that our study considers a single *cmp* function for a given App. For Apps requiring several computations, our leakage anal-

ysis still applies for each *cmp*, but the total leak can be accumulated across the set of functions. Also, the considered *cmp* does not allow parameters from the App (e.g., *cmp* is a similarity functions for time series or images having also an input parameter sent by the app). Parameters may introduce an additional attack channel allowing the attacker to increase the data leakage.

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

To reduce the potential data leakage, complementary security mechanisms can be employed for some Apps, e.g., imposing a query budget or limiting the σ predicates. Defining such restrictions and incorporating them into App manifests would definitely make sense as future work. Also, aggregate computations are generally basic and 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.

REFERENCES

[Anciaux et al., 2019] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Pucheral, P., Popa, I. S., and Scerri, G. (2019). Personal data management systems: The security and functionality standpoint. *Information Systems*, 80:13–35.

[Chaudhry et al., 2015] Chaudhry, A., Crowcroft, J., Howard, H., Madhavapeddy, A., Mortier, R., Haddadi, H., and McAuley, D. (2015). Personal Data: Thinking Inside the Box. *Aarhus Series on Human Centered Com.*

[Costan and Devadas, 2016] Costan, V. and Devadas, S. (2016). Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86.

[de Montjoye et al., 2014] de Montjoye, Y.-A., Shmueli, E., Wang, S. S., and Pentland, A. S. (2014). openPDS: Protecting the Privacy of Metadata through SafeAnswers. *PLoS one*, 9.

[Dwork, 2006] Dwork, C. (2006). Differential privacy. In *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*. Springer.

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

[Gentry, 2009] Gentry, C. (2009). *A Fully Homomorphic Encryption Scheme*. PhD thesis.

[Goltzsche et al., 2019] Goltzsche, D., Nieke, M., Knauth, T., and Kapitza, R. (2019). AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting. Middleware '19.

[Han and Hu, 2021] Han, Z. and Hu, H. (2021). ProDB: A memory-secure database using hardware enclave and practical oblivious RAM. *Information Systems*, 96:101681.

[Hunt et al., 2018] Hunt, T., Zhu, Z., Xu, Y., Peter, S., and Witchel, E. (2018). Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM TOCS*, 35(4).

[Myers, 1999] Myers, A. C. (1999). Jflow: Practical mostly-static information flow control. POPL '99. ACM.

[Novak et al., 2020] Novak, E., Aung, P. T., and Do, T. (2020). VPN+ Towards Detection and Remediation of Information Leakage on Smartphones. MDM '20.

[Pinto and Santos, 2019] Pinto, S. and Santos, N. (2019). Demystifying arm trustzone: A comprehensive survey. *ACM CSUR*, 51(6).

[Priebe et al., 2018] Priebe, C., Vaswani, K., and Costa, M. (2018). EnclaveDB: A Secure Database Using SGX. In *IEEE S&P*.

[Sabelfeld and Myers, 2003] Sabelfeld, A. and Myers, A. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19.

[Sambra et al., 2016] Sambra, A., Mansour, E., Hawke, S., Zereba, M., Greco, N., Ghanem, A., Zagidulin, D., Aboulnaga, A., and Berners-Lee, T. (2016). Solid: A platform for decentralized social applications based on linked data.

[Sun et al., 2016] Sun, M., Wei, T., and Lui, J. C. (2016). TaintART: A Practical Multi-Level Information-Flow Tracking System for Android RunTime. CCS.

[Weiser et al., 2019] Weiser, S., Mayr, L., Schwarz, M., and Gruss, D. (2019). SGXJail: defeating enclave malware via confinement. RAID.

[Zhou et al., 2021] Zhou, W., Cai, Y., Peng, Y., Wang, S., Ma, K., and Li, F. (2021). VeriDB: An SGX-Based Verifiable Database. SIGMOD.