

Trustworthy Distributed Computations on Personal Data Using Trusted Execution Environments

Riad Ladjel, Nicolas Anciaux, Philippe Pucheral, Guillaume Scerri

▶ **To cite this version:**

Riad Ladjel, Nicolas Anciaux, Philippe Pucheral, Guillaume Scerri. Trustworthy Distributed Computations on Personal Data Using Trusted Execution Environments. BDA 2019 - 35ème Conférence sur la Gestion de Données - Principes, Technologies et Applications, Oct 2019, Lyon, France. hal-02269211

HAL Id: hal-02269211
https://hal.science/hal-02269211

Submitted on 22 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Trustworthy Distributed Computations on Personal Data Using Trusted Execution Environments

Riad Ladjel
Inria, UVSQ, France
riad.ladjel@inria.fr

Nicolas Anciaux
Inria, UVSQ, France
nicolas.anciaux@inria.fr

Philippe Pucheral
Inria, UVSQ, France
philippe.pucheral@uvsq.fr

Guillaume Scerri
Inria, UVSQ, France
guillaume.scerri@uvsq.fr

Abstract— Thanks to new regulations like GDPR, Personal Data Management Systems (PDMS) have become a reality. This decentralized way of managing personal data provides a de facto protection against massive attacks on central servers. But, when performing distributed computations, this raises the question of how to preserve individuals' trust on their PDMS? And how to guarantee the integrity of the final result? This paper proposes a secure computing framework capitalizing on the use of Trusted Execution Environments at the edge of the network to tackle these questions.

Keywords— Data privacy; TEE; secure distributed computing

I. INTRODUCTION

Smart disclosure initiatives (e.g., Blue Button in the US, MiData in UK, MesInfos in France) and new privacy-protection regulations (e.g., GDPR in Europe [1]) allow individuals to get their personal data back and manage it under control, in a fully decentralized way, using so-called Personal Data Management Systems (PDMS) [3]. Decentralization is paramount in terms of privacy protection by reducing the Benefit/Cost ratio of an attack compared to a central server.

However, crossing data of multiple individuals (e.g., computing statistics or clustering data for an epidemiological or sociological study, training a neural network to organize bank records into categories or predict diagnoses according to medical symptoms, etc.) is of utmost personal and societal interest. This raises the question "how to preserve the trust of individuals on their PDMS while engaging their data in a distributed process that they cannot control?". The dual question from the querier side (i.e., the party initiating the processing) is "how to guarantee the honesty of a computation performed by a myriad of untrusted participants?". These are the two questions targeted by this paper.

Answering these questions requires establishing mutual trust between all parties in a distributed computation. On the one hand, any (PDMS) participant must get the guarantee that only the data required by the computation are collected and that only the final result of the computation he consents to contribute to, is disclosed (i.e., none of the collected raw data can be leaked). On the other hand, the querier must get the guarantee that the final result has been honestly computed, with the appropriate code, on top of genuine data. Besides this, the computing scheme must be generic and scalable (e.g., tens of thousands of participants) to have a practical interest.

No state of the art solution tackles all dimensions of this problem. Multi-party computation (MPC) works guarantee that only the final result of a computation is disclosed but they are either not generic in terms of supported computation or not scalable in the number of participants [10]. Similarly, gossip-based [2], homomorphic encryption-based [13] or differential privacy-based solutions are restricted to a limited set of operations that can be computed. Moreover, none of these solutions tackle the limited data collection and computation honesty issues. Recent works like [21] address the problem of authenticated query results, but focus on a single-user context, where the user is the querier.

In this paper, we argue that the emergence of Trusted Execution Environments (TEE) [15] definitely changes the game. TEEs, like ARM's TrustZone or Intel's Software Guard eXtention (SGX), are becoming omnipresent, from high-end servers to PC and mobile devices. TEEs are able to compute

arbitrary functions over sensitive data while guaranteeing data confidentiality and code integrity. This opens new opportunities to think about secure distributed processing with the hope to reconcile security with genericity and scalability.

But TEEs are far from providing a direct solution on their own. They have not been designed with edge computing involving a large number of participants in mind. Moreover, while TEE tamper-resistance makes attacks difficult, specific side-channel attacks have been shown feasible [19]. Without appropriate counter-measures, a minority of corrupted participants may endanger the data from the majority. Based on this statement, the paper makes four contributions:

- it defines a generic and scalable TEE-based computing protocol over decentralized PDMSs which provides the expected *mutual trust* and *computation honesty* properties, assuming this protocol has been safely executed;
- it provides, for each participant and the querier, a solution to *locally* check that the protocol has indeed been honestly executed, without resorting to any trusted third party;
- it proposes accurate counter-measures against side-channel attacks conducted by corrupted TEE participants;
- finally, it qualitatively and quantitatively evaluates the scalability and security of the solution on practical use-cases (group-by queries, k-means clustering).

II. PROBLEM FORMULATION

A. Security properties and limits of TEEs

Relying on secure hardware, existing Trusted Execution Environments (TEEs) provide three main security properties: (1) *code isolation*, meaning that an attacker controlling a corrupted user environment/OS cannot influence the behavior of a program executing within a TEE enclave, (2) *confidentiality*, meaning that private data residing in an enclave may never be observed, and (3) *attestation*, allowing to prove the identity of the code running inside a TEE [20].

The only type of attacks successfully conducted so far over TEE are side channel attacks [19]. The TEE in this case behaves in a "sealed glass proof" mode [18], i.e., the confidentiality property is compromised, but the isolation and attestation properties still holds (these properties are not challenged today). These attacks are complex to perform and require physically instrumenting the TEE, which prevents large scale attacks. However, TEEs corrupted by side-channel attacks cannot be detected by honest ones as their behavior is still the correct one.

B. Trust model

The trust model considered in this paper stems from the decentralized nature of the targeted infrastructure.

Untrusted user devices and infrastructure. No credible security assumptions can be made on the execution environment running on widely open personal devices (PC, laptop, home box, smartphone, etc.) managed by non-experts. We thus consider that the device OS and applications can be corrupted. We also consider the communication infrastructure as untrusted. We however assume that the communication flow incurred by the computed algorithm is (made) *data independent*, i.e., that personal data cannot be inferred by observing the communication pattern among participants[1].

Large set of trusted TEEs, small set of corrupted TEEs. We assume that each individual owns a TEE-enabled device hosting his personal data (i.e., his PDMS). This is definitely no longer fantasy considering the omnipresence of ARM's TrustZone or Intel's SGX on most PC, tablets and smartphones. As explained above, a small subset of TEEs could have been corrupted by malicious participants to break their confidentiality with side-channel attacks.

Trusted computation code. We consider that the code distributed to the participants has been carefully reviewed and approved beforehand by a regulatory body (e.g., an association or national privacy regulatory agency). But the

[1] Scalable solutions exist to meet this requirement [11] but this problem is orthogonal to this paper.

fact that the code is trusted does not imply that its execution behaves as expected.

Trusted citizen identity. We consider that citizens have been assigned a private/public key by a trusted (e.g., governmental) entity (e.g., as used today for paying taxes online). This prohibits attackers generating multiple identities with the objective to massively contribute to a computation to isolate a small set of participants and infer their data.

C. Problem statement

The problem can be formulated as follows: how to translate the trust provided to the computation code by the regulatory body into a mutual trust between all parties participating to the computation under the presented trust model? To solve this problem, the following properties need to be satisfied:

Mutual trust. Assuming that the declared code is executed within TEEs, mutual trust guarantees that: (1) only the final result *r* of the computation can be disclosed, i.e., none of the raw data of any participant is leaked and *r* is honestly computed as declared, (2) only the data strictly specified for the computation is requested from the participant PDMSs, (3) the computation code is generic and makes it possible to verify that any collected data is genuine[2].

Local assurance of validity. The querier and each involved participant must be able to monitor *locally* (i.e., on its own, without relying on a central trusted party) that the computation is being performed in compliance with the code declaration, by *all* other participants. If any honest participant detects a validity violation, an error is produced and the computation stops without producing any other (partial) result.

Resilience to side-channel attacks. Assuming a small fraction of malicious (colluding) participants involved in the computation with corrupted TEEs, our framework must (1) guarantee that the leakage remains circumscribed to the data manipulated by the sole corrupted TEEs, (2) prevent the attackers from targeting a specific intermediate result (e.g., sensitive data or data of targeted participants) and (3) maximize the Cost/Benefit ratio of an attack. Note that this is the best we can do assuming that the code manipulates clear data and that side channel attacks can be performed. In addition, the means to achieve resilience should maintain the communication flow independent of the data being processed (i.e., attack resiliency should not affect the *data independence* assumption made in our trust model).

To have a practical interest, the solution must finally: (1) be generic enough to support any distributed computations (e.g., from simple aggregate queries to advanced machine learning computations) and (2) scale to a large population (e.g., tens of thousands) of individuals.

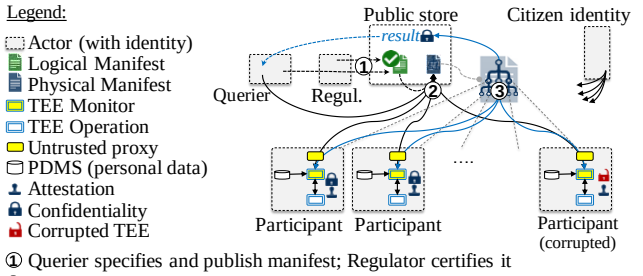


Figure 1 Manifest-based distributed computation.

III. MUTUAL TRUST

To provide the *mutual trust* property, we propose adopting a manifest-based approach. As described in Fig. 1, this approach is conducted in three steps:

Step1: logical manifest declaration. We call *Querier* an entity (e.g., a research lab, a statistic agency or a company, acting as a data controller in the GDPR sense) wishing to execute a treatment over personal data. The Querier specifies a *Logical Manifest* describing the computation to be performed, namely: its purpose, the source code of the operator to be run at each participant, the distributed execution plan materializing the data flow between operators and a set

[2] Assuming data genuineness can be actually verified by the running code in any way (e.g., thanks to a digital signature).

of privacy rules to be fulfilled, including data collection rules and expected number of participants. The Querier submits this logical manifest to a Regulatory body which certifies its compliance with the expected privacy practices. The certified logical manifest is then published in a public manifest store where it can be downloaded by individuals wishing to participate. We provide below an example (deliberately naïve for the sake of simplicity) of a logical manifest for a group-by query implemented using a MapReduce-like framework.

Example 1: 'Group-by' manifest.

```
Purpose:
  Compute the mean quantity of anxiolytic
  prescribed to employees group by employer
Operators:
  mapper source code
  reducer source code
Distributed execution plan and dataflow:
  Number of mappers: 10.000
  Number of reducers: 100
  Any mapper linked to all reducers
Collection rules:
  SELECT employer_name FROM Job;
  SELECT sum(qty)FROM Presc
  WHERE drugtype = 'anxiolytic';
Number of participants (data collectors): 10.000
Querier Public key: Rex2%ÃźHj6k7âĂę
```

Step2: physical manifest construction. Once certified, the manifest can be viewed as a logical distributed query plan (participants are not yet identified). When a sufficient number of potential participants consent to contribute with their data, a *Physical Manifest* is collectively established by the TEEs of all participants (according to our trust model, each participant is equipped with a TEE). A physical manifest assigns an operator to each participant. As detailed in Section V, this step is critical for *resilience to side-channel attacks*, by prohibiting corrupted participants from selecting specific operators in the query plan for malicious purpose.

Step3: physical manifest evaluation. Each participant downloads the physical manifest (or the subpart allocated to him). The participant's TEE initializes an enclave to execute his assigned operator and establishes communication channels with the TEEs of other participants supposed to exchange data with him (according to the manifest distributed execution plan). The participants then contributes his personal data to the operator and allows the computation to proceed. Once all participants have executed their task, the end-result is delivered to the querier.

Let us introduce the following definitions in order to analyze how *mutual trust* is achieved.

Definition 1: Distributed Execution Plan (DEP). A distributed execution plan DEP is defined as a directed graph (V, E) where V vertices are couples $(op_i, a_j) \in OP \times A$ with OP the set of operators to be computed and A the set of computing agents, and E edges are couples $(<op_i, a_j>, <op_k, a_l>)$ materializing the dataflow among operators, namely the transmission by a_j to a_l of op_i output. For any $v_i \in V$, we denote by $Ant(v_i)$ (resp. $Succ(v_i)$) the antecedents (resp. successors) of v_i in the DEP, that is the vertices linked to v_i by a direct incoming (resp. outgoing) edge.

This representation of distributed execution plans is generic enough to capture most distributed data-oriented computations. Based on this definition, we can introduce the notion of logical manifest.

Definition 2: Logical Manifest (LM). A logical manifest *LM* is defined as a tuple $<PU, DEP, CR, N>$, with *PU* the textual purpose declaration, *DEP* a distributed execution plan, *CR* the collection rule applied at each participant and *N* the expected number of participants.

The *CR* declaration translates the limited collection principle enacted in all legislations protecting data privacy (i.e., no data other than the ones strictly necessary to reach the declared purpose *PU* will be collected). We assume that this declaration is done using a basic assertional language (e.g., a subset of an SQL-like language) easily interpretable by the Regulatory body on one side and easily translatable into the specific query language of any PDMSs on the participant's side. For the sake of simplicity, we assume that the data queried at each participant follow the same schema (if it is not the case, it is basically a matter of translating the collection rules in different schemas). *N* plays a dual role: it represents both a significance threshold for the Querier wrt. the declared

purpose and a privacy threshold for the Regulatory body wrt. the risk of reidentification of any individual in the final result.

The notion of physical manifest can be defined as follows.

Definition 3: Physical Manifest (PM). A physical manifest *PM* is defined as a tuple $<LM, P, F, Q_{CR}>$ such that: (1) function F: $LM.DEP.A \rightarrow P$ assigns agents to the participants *P* contributing to the computation of *LM*; (2) *F* is bijective, so that a given participant cannot play the role of different agents and each agent is represented by a participant; (3) any query $q_i \in Q_{CR}$ is the translation for participant p_j of the collection rule *LM.CR* into the query language of his PDMS.

Definition 4: PM valid execution. An execution of a physical manifest *PM* is said valid if the execution has not deviated in any manner from what is specified in *LM*, i.e., (i) the operators in *LM.DEP.OP* are each executed by the TEE of the participant designated by *F* while respecting the dataflow imposed by *LM.DEP.E*, (ii) the TEE of any participant p_i queries its host with q_i, (iii) *N* different participants contribute to the computation and (iv) all data exchanged between the participants' TEEs are encrypted with session keys.

Lemma 1. Under the hypothesis *H1* that the execution of a *PM* is valid and *H2* that no TEE have been corrupted, the *mutual trust* property is satisfied.

We postpone to Section IV how to achieve hypothesis *H1* and to Section V the counter-measures suggested in the case hypothesis *H2* does not hold.

Proof of Lemma 1. The three conditions in *mutual trust* definition given in Section II hold by construction. First, condition (1) is satisfied because *H1* guarantees that each operator in *DEP.OP* is executed within a TEE, and *H2* and the TEE's confidentiality property ensure that no data can leak other than the input and output of each *DEP.OP*. Encrypting the data exchanges between each vertex v_i and $Ant(v_i)$ and $Succ(v_i)$ in *DEP* with a session key ensures the confidentiality of the global execution of *PM.DEP*. The final result is itself sent encrypted to the Querier so that no raw data other than the final result can leak all along the execution. Second, condition (2) stems from the fact that each participant p_i is presented with q_i which is a translation of *LM.CR*. The honest execution of q_i over p_i's PDMS remains however under the participant's responsibility who selected it to protect his personal data. Regarding condition (3), *H1* and *H2* again guarantee the integrity of the global execution of *PM.DEP*. Note that this guarantee holds even in the presence of corrupted TEEs since side-channel attacks on TEEs may compromise the confidentiality of the processing but not the isolation property. It immediately follows that any check integrated in the operator code can be faithfully performed on cleartext data, thus ensuring genericity.

Compared to state of the art solutions, our manifest-based approach holds the capacity to reconcile security with genericity and scalability. First, the TEE confidentiality property can be leveraged to execute the computation code at each participant over cleartext genuine data. Second, the shape of the DEP and then the resulting number of messages exchanged among participants, directly results from the distributed computation to be performed. Hence, conversely to MPC, homomorphic encryption, Gossip or Differential privacy approaches, no computational constraints compromising genericity nor performance constraints compromising scalability need to be introduced in the processing for security reasons.

IV. LOCAL ASSURANCE OF VALIDITY

Once mutual trust is ensured, one needs to ensure that each participant gets the assurance that the computation was performed as expected. Ideally, this means that the computation should behave as if all participants could continuously monitor all the others, i.e., check all operator computations, ensuring correction of sent/received data at each step, and abort the whole process if any misbehavior happens. This is formalized in the following definition. At this stage, we assume that the execution plan has been produced by an arbitrary function *build_phys_manifest*, assigning a position *i* in the execution plan to each participant (the strategy

for performing this assignment is discussed in Section V). We also assume that the local code executed by a participant either terminates successfully or explicitly returns an error.

Definition 5: *locally checkable execution*. The execution of a distributed execution plan DEP is said locally checkable if for any participant $p_j \in PM.P$, either (i) p_j's view of the partial execution up to p_j's role is valid or (ii) p_j returns an error and no data is ever transmitted to other participants.

An immediate consequence of Definition 5 is that, for any locally checkable execution, either a global result is produced if the execution is valid or no intermediate values is ever leaked. It follows that a protocol guaranteeing locally checkable executions for a DEP exactly provides local assurance of validity as any deviation from the normal execution would result in an invalid execution and would therefore result in an error at the participant's level.

As participants execute code in TEEs, a naïve way to satisfy Definition 5 is to instrument the code of each operator in order to make sure that before sending out any (partial) result the code gets approval from all other participants. While this solution trivially satisfies our goal of local assurance of validity, the communication overhead with a large number of participants is overwhelming.

In order to overcome the aforementioned problem, we leverage the fact that using the TEE mechanisms and attestation, one can rely on checks made within other participant's TEEs. In our architecture, the foundation of local checkability is the decomposition of the code running at each participant in a generic TEE monitor and a specific TEE computation code. The objective of this distinction is to avoid the need for any participant to recompile the code running on the other participants and compute its hash to evaluate the validity of the requested remote attestations. The execution at each participant then works as follows: (1) untrusted code executed on the local host, called untrusted proxy in Fig. 2, creates a TEE enclave and launches the TEE monitor code inside this enclave, (2) the TEE monitor, the role of which is to interpret the manifest and drive the local execution, creates a second enclave to launch the TEE computation code corresponding to the operator assigned to the participant in the execution plan. Note that all of the scheduling is performed by the untrusted proxy, in particular waking up TEE monitors as they are needed for the computation.

The TEE monitor code is identical for each participant, so that its hash is known by everyone. This code is minimal, can be easily formally proved and is assumed trusted by all participants. This lets us consider the manifest *LM* as data, including the code of the local operator to be computed, let each local TEE monitor check the integrity of this data and then attest the other participants (antecedents and successors in the execution plan) to the genuineness of the TEE computation code. Antecedents and successors can easily check in turn the validity of the received remote attestation by checking only the genuineness of the remote TEE monitor. This double attestation by the antecedents and by the successors is mandatory to guarantee, for each participant, the validity of the inputs it receives and the authenticity of the recipients for its own outputs. This transitive attestation principle is depicted in Fig. 2.

Following this strategy, local checkability is guaranteed. Intuitively, if a specific participant does not execute the genuine TEE monitor, it will be unable to provide a valid attestation to its partners (antecedents/successors) which will stop the execution and return an error. Then, if all participants run the correct TEE monitor and execute the same manifest, the execution is necessarily correct, since the TEE monitor only executes its dedicated code, and attestation prevents attacks from the OS on the result of the TEE computation code. If, however, one participant does not execute the correct manifest, its antecedents/successors will fail during the manifest verification. Finally, for any execution plan represented by a connected graph, the validity of the global execution is obtained by propagating errors through the execution graph, if an error occurs at any point during the

computation. In order to prevent an attacker from running a large number of instances of a computation code in enclaves, each enclave must be tied to an identity, certified by a *citizen identity provider*.

The pseudo code of the TEE monitor is provided in Algorithm 1. For the sake of conciseness, we restrict this algorithm to the management of tree-based execution plans, however extending it to any graph is just a matter of allowing multiple successors. Note that the scheduling of the execution and errors propagation can be handled by untrusted code. Indeed, if a participant encounters an error, it would typically propagate it upstream so as not to let successor's enclaves hanging. However, it is by no means security critical as successor's enclave would simply never execute if they fail to receive their antecedents' inputs.

Algorithm 1. TEE monitor

Input: *LM* the logical manifest, $id = (sk_i, pk_i, cert_i)$ the participants' cryptographic identity and the corresponding certificate
Output: *boolean* indicating success

1. **if** *verify*(*LM*) = *false* **then** // verify manifest
2. **return** *error*
3. $PM \leftarrow build_phys_manifest(LM, id)$ // build physical manifest
4. $i \leftarrow get_my_position(PM, sk_i)$
5. $PM_i \leftarrow extract(PM, i)$
6. $Q_i, P_i, C_i, op_i \leftarrow Parse(PM_i)$
7. **for each** *antecedent* $\in Ci$ **do** // get antecedents' outputs
8. **if** *not*(*channel*(*antecedent*, *self.code*)) **then return** *error*
9. **if** *not*(*id_check*(*antecedent*)) **then return** *error*
10. **if** *antecedent.PM* $\neq$ *PM* **then return** *error*
11. *input_tuples* +$\leftarrow$ *accept_input*(*antecedent*)
12. *input_tuples* +$\leftarrow$ *out_call*(*Qi*) // query PDMS
13. $E_{OPi} \leftarrow create_enclave(op_i)$ // create op_i enclave
14. **if** *not*(*channel*(E_{OPi}, op_i) **then return** *error*
15. *send_tuples*(*input_tuples*, E_{OPi}) // produce output
/* *NB*: *integrity of input_tuple is checked in OPi enclave code* */
16. $res_op_i \leftarrow accept_input(E_{OPi})$ // execute op_i
17. *successor* $\leftarrow$ *get_ successor(PM,*res_op_i*)*
18. **if** *successor* = *querier* **then**
19. $a \leftarrow attest(res_op_i, PM)$
20. *send*(*a*, *res_opi*)
21. **return** *success*
22. **if** *not*(*channel*(*successor*), *self.code*) **then return** *error*
23. **if** *not*(*id_check*(*successor*)) **then return** *error*
24. **if** *successor.PM* $\neq$ *PM* **then return** *error*
25. *send_tuples*(res_op_i, *successor*)
26. **return** *success*

While hidden in the pseudo code, we assume that all communications between participants and the different enclaves are performed on secure channels. This is crucial to ensure that the endpoints of channels lie in real TEE enclaves and to prevent an adversary capable of observing the communications from getting access to user data. A primitive reaching this goal is called attested key exchange [5]. It allows to exchange a key with an enclave executing a specific program, and hence ensures (using the attestation mechanism) that the endpoint of the channel lies within an enclave and that the enclave is executing the expected program, even if the administrator of the machine running the enclave is corrupted. We abstract this creation of a secure channel as *channel*(*remote*, *expected_code*) where *remote* is the remote enclave and *expected_code* is the code expected to be running in the remote enclave. The cost is essentially *1* remote attestation and *2* communications. Once established, all communications are assumed to be done on this channel. For simplicity's sake we abstract away who is the initiator of the secure channel and view this process as symmetric.

Algorithm description. In lines 1 to 6, the integrity of the logical manifest is verified by checking its signature, the physical manifest is built in collaboration with the other participating TEE monitors (cf. Section V, which also covers the explanation of line 4, not required in this section) and the part of the manifest related to this participant is extracted (i.e., the set of its antecedents/successors, the data collection query used to retrieve data from the local PDMS and the code of the operator to be evaluated locally).

Then, in lines 7 to 12, the attestation of each antecedent is verified, by comparing the hash value of the code it is running to the hash value of the TEE monitor code (common to each participant). Once the antecedent TEE monitor is known to be correct, we check that it runs the correct manifest. We also check its identity by requiring its enclave to send it. This provides enough assurance because once we know the code of its enclave we know that it will honestly send its identity. Finally, the input tuples of the local operator are retrieved from its antecedents and/or the local PDMS of this participant.

In lines 13 to 16, the TEE monitor creates an additional enclave for the operator to be run (its code is part of the

manifest) and requests an attestation from this enclave (the hash of the operator is compared to the hash of the code computed by the TEE monitor) to make sure that the host did not compromise or impersonate the operator code. Then the monitor establishes a secure channel with the operator enclave, using an attested key exchange as in [5] and TEE monitor calls the operator using the appropriate inputs.

Finally, in lines 17 to 26, the TEE monitor, either sends the result to the querier if its result is the final result, together with an attestation guaranteeing the result was indeed produced by the correct computation of the specified data; or sends its result to the next participants as planned by the DEP.

Proposition 1. Algorithm 1 satisfies the locally checkable execution property for the physical manifest *PM* derived from the logical manifest *LM* by the *build_phys_manifest* function.

The sketch of proof of Proposition 1 is given in [23].

V. Resilience to Side-Channel Attacks

According to our trust model, a small fraction of TEEs can be instrumented by malicious (colluding) participants owning them to conduct side-channel attacks compromising the TEE *confidentiality* property. This issue is paramount in our Manifest-based approach which draws its genericity and scalability from the fact that computing nodes manipulate cleartext genuine data, putting them at risk.

The *resilience to side-channel attacks* property introduced in Section II, states first that the leakage generated by an attack must be circumscribed to the data manipulated solely by the corrupted TEEs. This is intrinsically achieved in our proposal by never sharing any cryptographic information among different nodes. A second requirement is to prevent any attacker from targeting specific personal data. *Randomness* and *Sampling* are introduced next to achieve this goal. Finally, DEP *reshaping* is proposed to tackle the third requirement, i.e., maximizing the average Cost/Benefit ratio of an attack.

A. Randomness and Sampling

In a physical manifest, we distinguish participants assigned to a collection task (which contribute to the query with their own personal raw data) from participants assigned to a computation task (which process personal data produced by other participants). Attacking any TEE running a collection task has no interest since the attacker only gains access to his own personal data. Hence, the primary objective of an attacker is to tamper with the building phase of a physical manifest such that his TEE is assigned a computation task to leak the data it manipulates. The *randomness* counter-measure assigns a random position in the DEP to each participant. More precisely, it ensures that for a given physical manifest *PM*, any participant $p_j \in PM.P$ is able to locally verify that its position and the position of any other participant in *PM.P* have been obtained randomly. A protocol achieving this goal, adapted from [22], can be found in [23]. The idea is to ensure that the randomness is generated by an enclave once the querier has committed to the list of participants. As we consider that enclave integrity may not be compromised, this ensures that this number is chosen uniformly at random even in the presence of an active adversary.

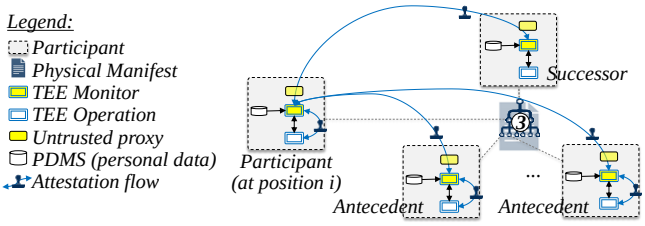


Figure 2. Attestation flow for position i.

The second counter-measure to prevent an attacker from selecting its position in the DEP is to add a *sampling* phase in *Step2* (physical manifest construction, Section III) by selecting a given rate σ of individuals accepting to contribute to the computation. The lower this rate σ, the more TEEs need be corrupted to keep the same probability of selecting a given position in the built DEP for an attacker. For conciseness, we

refer to a technical report [23] for details on *randomness* and *sampling* algorithms and security proofs.

B. DEP Reshaping

In our context, the *Cost* factor of the *Cost/Benefit* ratio is expressed in terms of the number of TEEs to corrupt and the *Benefit* is measured by the amount of personal data leaked by the attack. While *randomness* and *sampling* contribute to exacerbate the *Cost* factor, our third countermeasure aims at reducing the amount of raw data exposed at a single TEE. To introduce the idea, let us consider a DEP with *n* participants, among which *m* computation nodes computing a function *f* and *n-m* collection nodes contributing with their own data. With the *randomness* countermeasure, the probability to corrupt exactly *t* computation nodes among *m* with *c* corrupted participants (side-channel attacks), follows an hypergeometric distribution $p_{n,m,c}(x = t) = \binom{m}{x}\binom{n-m}{c-x}/\binom{n}{c}$. The probability of corrupting *t* or more computation tasks over *m* is then:

$$p_{n,m,c}(t \le x \le m) = \sum_{x=t}^{m} \binom{m}{x}\binom{n-m}{c-x}/\binom{n}{c}.$$

With *n=10000*, *m=10* and *c=100* (which is a high number of corrupted participants), the probability of corrupting at least one computation node is $p_{10000,10,100}(1 \le x \le 10)=0.095$, while with *m=100*, this probability drops to $p_{10000,100,100}(10 \le x \le 100)=5.10^{-8}$. For simplicity, we assume that each participant contributes with exactly one tuple mapped to a single computation node, hence each computation node processes (and may endanger) on average *N*/*m* tuples (*1000* tuples here).

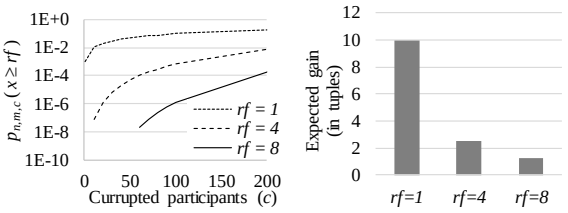


Figure 3. Probability and expected values in tuples of successful attacks.

Fig. 3 (left) plots the privacy benefit of increasing the number of computation nodes by reshaping the DEP such that each initial computation node m_i is split in *rf* new computation nodes sharing m_i's initial computing load, with *rf* denoting the *reshaping factor*. More precisely, this curve plots the probability to leak the same amount of data as with the original settings in function of the number *c* of corrupted nodes with different reshaping factor *rf* (e.g., *rf=1* is the initial settings with *m=10* computation nodes, *rf*=2 means *m=20*, etc.). Unsurprisingly, increasing *rf* dramatically decreases the probability of an attack leaking the same amount of tuples since *rf* different computation nodes (among *rf*×*m*) must now be corrupted with the same number *c* of corrupted participants.

Fig. 3 (right) shows the expected value in number of leaked tuples (i.e., sum of the probability of a successful attack on some computation nodes times the number of tuples leaked) in the case of successful side-channel attacks with *c=100* corrupted nodes. The expected gain is always small, although the number *c* of corrupted TEE is relatively high, and reduces linearly with *rf*, which is deterrent for attackers. Indeed, the probability to successfully break two computation nodes is close to zero, hence the expected gain is nearly given by the probability of breaking a single computation node times the number of leaked tuples processed in that node, which linearly decreases with *rf*. These results are true if *m* and *c* are small compared to *n*, which is typically the case in our context.

The conclusion is that, while maximizing the distribution of a computation has recognized virtues in terms of performance and scalability (explaining the success of MapReduce or Spark models), this strategy leads as well to a better resilience against side-channel attacks. Maximizing the distribution can be done by exploiting some properties of the functions to be evaluated by *DEP* computation nodes:

Definition 6: *Distributive function*. Let *f* be a function to be computed over a dataset *D*, *f* is said distributive if there exists a function *g* such that $f(D) = g(f(D_1), f(D_2), \ldots, f(D_N))$ where D_i forms a partition of *D* (e.g., $D = \cup_i (\sigma(i, D_i))$ with σ a selection function).

Definition 7: *Algebraic function*. A function f is said algebraic if f can be computed by a combination of distributive functions (e.g., *mean*(*D*) = *sum*(*D*)/*count*(*D*)).

For any DEP node computing a distributive or algebraic function, the number of *D* input tuples exposed to that node can be linearly reduced by augmenting the number of D_i partitions in the same proportion. This general principle, called DEP *reshaping*, splits distributive/algebraic tasks allocated to a single participant into several tasks allocated to different nodes, each working on a partition of the initial input.

Definition 8: *rf-reshaping*. Given an attribution function $at: V \rightarrow \{1, \ldots, rf\}$ associating vertices to integers uniformly, a distributed execution plan *DEP*'(*V*',*E*') is obtained by *rf-reshaping* from *DEP*(*V*,*E*) such that: $V' \supseteq V$ and $\forall v_i=(a_i,op_i) \in V /$ distrib_algebra(op_i)=true $\Rightarrow$ $v_{i,j}'=(a_{i,j},op_i) \in V'$ with *j*:*1*..*rf*, and $v_{i,j}' \in$ Ant(v_i) in E' and $\forall v \in$ Ant(v_i) in E, $v \in$ Ant($v_{i,j}'$) with $j = at(v)$ in E'.

This definition is illustrated on Fig. 4, showing a DEP with *6* additional computation nodes obtained by *3-reshaping* from an initial DEP with only *2* computation nodes. Note that the communication overhead is very small, as only one additional message from each added reshaped node to the original node is produced compared to the original execution. In the remainder of the paper, we call a cluster all vertices attributed to the same reshaped node (i.e. $at^{-1}(j)$).

All other things being equal, *rf*-reshaping drastically reduce the data exposure at each computing node. Indeed, for any distributive/algebraic vertex v_i in DEP, *rf*-reshaping divides the probability of gaining access to the entire input *D* of v_i by a factor $\binom{n-rf}{c-rf} \prod_{i=1..rf} (n-i)/(c-i)$.

The final issue is showing that *rf*-reshaping may hurt the independence between the processed data and the dataflow as specified in the initial DEP. Recall that a communication flow *E* is said *data independent* if the DEP is such that personal data cannot be inferred from observing the communication pattern among participants. *E* can be *data independent* by construction (e.g., broadcast-based algorithm) or be made *data independent* for privacy concern (e.g., sending fake data among participants to normalize the communications). It is thus mandatory to preserve this independence.

Lemma 2. If the communication flow *E* of a distributed execution plan *DEP*(*V*,*E*) is *data independent*, the communication flow *E*' of any *DEP*'(*V*',*E*') obtained by *rf-reshaping* of *DEP*(*V*,*E*) is also *data independent.*

The result is ensured by the fact that the communication flow in the *DEP'* only depends on the communication pattern in *DEP* and the *at*( ) function in Definition 8, which in turn only depends on vertices identifiers and not on data. Hence, the communication flow E' reveals nothing more about the transmitted data compared to E.

The *rf*-reshaping principle can be applied in many practical examples of computations over distributed PDMSs, ranging from simple statistical queries to big data analysis, as illustrated in the validation section. The *rf*-reshaping process can be automatically performed by a precompiler taking as input a logical manifest *LM* and producing a transformed logical manifest LM_{minExp} minimizing data exposure for the participants for each node computing distributive or algebraic functions. The degree of distribution impacts the performance and the protection of raw data in case of successful attacks (see Section VI), but selecting the optimal strategy and integrating it into a precompiler is left for future work.

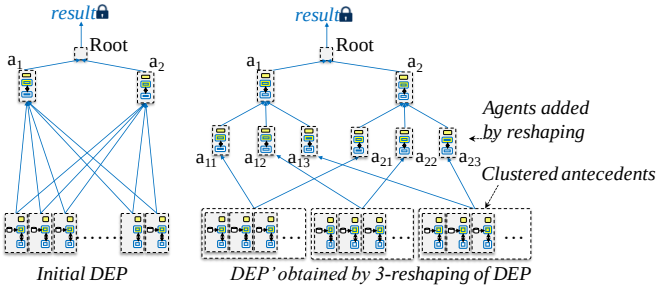
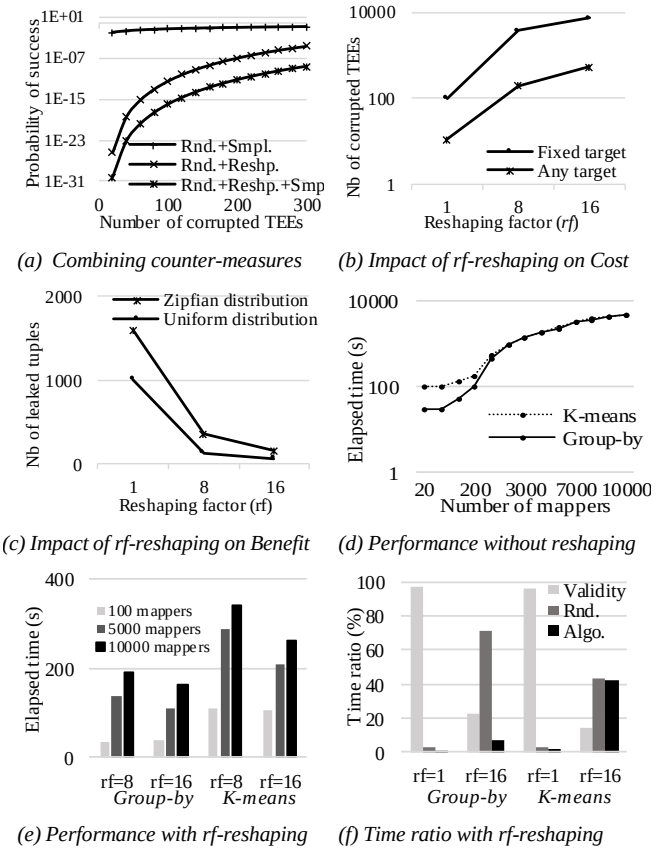


Figure 4. 3-reshaping of DEP with m=2 distributive computation nodes.

VI. VALIDATION

The effectiveness of the solution in terms of security and scalability is assessed on a platform composed of 8 SGX capable machines with Intel I5-7200U and 16GB RAM,

equipped with Intel SGX SDK 2.3.101 over Ubuntu 16.04. We consider two use-cases of distributed processing over personal data, to validate the genericity of our framework.



(a) Combining counter-measures *(b) Impact of rf-reshaping on Cost*

(c) Impact of rf-reshaping on Benefit *(d) Performance without reshaping*

(e) Performance with rf-reshaping *(f) Time ratio with rf-reshaping*

Figure 5. Security and performance evaluation.

Group-by aggregation. We consider a MapReduce-like implementation of an aggregate with a group by query run over distributed PDMSs. The processing is as follows: (1) each mapper sends a couple (h(*group_key*), *value*) to a reducer where h is a hash function which projects the group key on a given reducer, (2) each reducer computes the aggregate function over the values received for the group keys it manages. If the aggregate function is distributive (e.g., count, min, max, sum, rank, etc.) or algebraic (e.g., avg, var, etc.), *rf*-reshaping is applied to all reducer nodes. Sub-reducer nodes contribute to the computation of the function for a subset of a grouping value and the initial reducers combine their work.

K-means clustering. It is similarly computed: (1) k initial means representing the centroid of k clusters are randomly generated by the querier and sent to all participants to initialize the processing, (2) each participant acting as a mapper computes its distance with these k means and sends back its data to the reducer node managing the closest cluster, (3) each reducer recomputes the new centroid of the cluster it manages based on the data received from the mappers and sends it back to all participants. Steps 2 and 3 are repeated a number of times. The function computed by step 3 is *algebraic* since the centroid of a cluster c_i can be computed thanks to *sums* and *counts* computed over all sub-clusters of c_i. Hence, the number of reducers in step 3 can also be arbitrarily augmented by *rf*-reshaping, such that each of the k initial reducers is preceded in *DEP* by a set of sub-reducers computing a partial centroid.

The SGX platform is used to perform real measurements at small scale (up to *100* participants) and to calibrate an analytical model required to conduct large scale experiments. We used synthetic datasets since the primary goal is not studying the peak performance for specific data distributions and save seconds or minutes (manual surveys over thousands of participants usually take weeks). We present in Fig. 5 a summary of our results and give the main outcome. We refer to [23] for extensive experiments and details.

Several conclusions can be drawn. First, even if the overall time can be considered rather high (tens of minutes for large numbers of participants), it is not a critical issue in our context from a querier perspective. Second, *rf*-reshaping drastically reduces the elapsed time by augmenting the parallelism while decreasing the number of attestations between mappers and reducers. Third *rf*-reshaping is deterrent for attackers by acting on the Benefit/Cost ratio, even for low values of *rf*.

VII. RELATED WORKS

Several works focus on ***protecting outsourced databases*** with encryption, but most of the existing encryption schemes applied to databases have been shown vulnerable to inference attacks [7]. Going further induces fully homomorphic

encryption [8] with intractable performance issues. Some works [4] deploy secure hardware at database server side. These solutions are centralized by nature and do not match our assumption that no hardware module is perfectly secure.

In terms of decentralized computing, Secure Multi-Party Computations (MPC) have been adapted to databases (e.g. SMCQL [6]), but only support a few tens of participants. Using secure hardware, TrustedPals [9,12] makes MPC more scalable, but their goal and security assumptions differ from ours. TrustedPals ensures that results are distributed to all honest parties, and that all received results are consistent. Hence, TrustedPals is highly fault tolerant as opposed to our solution. Our solution is simpler, as it only aims at delivering the result to a querier, and thus does away with the consensus protocols. However, at the end of the computation, all involved parties in TrustedPals hold the entirety of the data within their security module, which would be unacceptable in our case as some TEEs might be compromised.

Several works suggest distributed computation schemes providing anonymous data exchanges and confidential processing using gossip-style protocols [2]. They typically scale well but are not generic in terms of computations. Similarly, decentralized processing solutions based on secure hardware have also been proposed for aggregate queries [17] but do not match our genericity objective.

To the best of our knowledge, all works regarding executing data oriented task using SGX (e.g. Ryoan [14]) have a unique controller, as opposed to our setting where no unique individual is supposed to be in control of the computation. Additionally, most of the time this controller also provides the data to be computed on. This greatly simplifies the problem as a same controller verifies all enclaves and organizes the computation. Other works [14], following VC3 [16], provide SGX-based Map-Reduce frameworks for executing computations in the cloud to ensure data confidentiality and hide communication patterns between mappers and reducers, but again, they consider a unique controller. Communication between enclaves in these works is quite similar to ours, but the assumption of a unique controller completely removes the need for establishing trust at a local level, which is exactly what our notion of local checkability aims at solving.

VIII. Conclusion

Smart disclosure initiatives and new regulations push for the adoption of Personal Data Management Systems managed under individual's control while keeping the capability to cross personal data of multiple individuals (e.g., economic, epidemiological or sociological studies). However, without appropriate security measures, the risk is high to see individuals refuse their contribution. Only fragmented solutions have emerged so far. The generalization of Trusted Execution Environment at the edge of the network changes the game. This paper capitalizes on this trend and proposes a generic secure decentralized computing framework where each participant gains the assurance that his data is used for the purpose he consents to and that only the final result is disclosed. Conversely, the querier is assured that the result has been honestly computed. We have shown the practicality of the solution in terms of privacy and performance. We hope that this work will lay the groundwork for thinking differently about decentralized computing on personal data.

Acknowledgements

This research is supported by the ANR PerSoCloud grant no ANR-16-CE39-0014.

References

[1] European Parliament. General Data Protection Regulation. Law. (27 April 2016). https://gdpr-info.eu/

[2] T. Allard, G. Hébrail, E. Pacitti, et al. Chiaroscuro: Transparency and privacy for massive personal time-series clustering. SIGMOD, 2015.

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

[4] A. Arasu and R. Kaushik. Oblivious query processing. In ICDT, 2014.

[5] M. Barbosa, B. Portela, G. Scerri, et al. Foundations of hardware based attested computation and application to SGX. In EuroS&P, 2016.

[6] J. Bater, G. Elliott, C. Eggen, et al, and J. Rogers. SMCQL: secure query processing for private data networks. PVLDB, *10*(6), 2017.

[7] V. Bindschaedler, P. Grubbs, D. Cash, et al. The tao of inference in privacy-protected databases. PVLDB, *11*(11), 2018.

[8] D. Boneh, C. Gentry, S. Halevi, F.Wang, and D. J.Wu. Private database queries using somewhat homomorphic encryption. ACNS, 2013.

[9] R. Cortiñas et al. Secure Failure Detection and Consensus in TrustedPals. IEEE Trans. Dependable Sec. Comput. *9*(4), 2012.

[10] I. Damgård, M. Keller, E. Larraia, et al. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. ESORICS, 2013.

[11] A. Dinh, P. Saxena, C. Chang, et al. M2R: enabling strong-er privacy in mapreduce computation. USENIX Security Symposium, 2015.

[12] M. Fort, F.C. Freiling, L.D. Penso et al. TrustedPals: Secure Multiparty Computation Implemented with Smart Cards. ESORICS, 2006.

[13] T. Ge and S. B. Zdonik. Answering aggregation queries in a secure system model. VLDB, 2007.

[14] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. OSDI, 2016.

[15] M. Sabt, M. Achemlal, et al. Trusted execution environment: What it is, and what it is not. TrustCom/BigDataSE/ISPA (1), 2015.

[16] F. Schuster, M. Costa, C. Fournet, et al. VC3: trustworthy data analytics in the cloud using SGX. S&P, 2015.

[17] Q. To, B. Nguyen, P. Pucheral. Private and scalable execution of SQL aggregates on a secure decentralized architecture. TODS, *41*(3), 2016.

[18] F. Tramèr, F. Zhang, H. Lin, et al. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. EuroS&P, 2017.

[19] W. Wang, G. Chen, X. Pan, et al. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. CCS, 2017.

[20] I. Anati, S. Gueron, S. Johnson, et al. Innovative technology for CPU based attestation and sealing. HASP, 2013.

[21] Zhang, Y., Genkin, D., Katz, J., et al. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. S&P, 2017.

[22] M. Backes, et al. CSAR. A Practical and Provable Technique to Make Randomized Systems Accountable. In NDSS, vol. 9. 2009.

[23] R. Ladjel, et al. Trustworthy Distributed Computations on Personal Data. Inria report. http://petrus.inria.fr/~anciaux/papers/TR.pdf, 2019.