Using Differential Privacy to Efficiently Mitigate Side Channels in Distributed Analytics

Min Xu*
University of Chicago
xum@cs.uchicago.edu

Antonis Papadimitriou*
University of Pennsylvania
antonis.papadimitriou@gmail.com

Ariel Feldman
University of Chicago
arielfeldman@cs.uchicago.edu

Andreas Haeberlen
University of Pennsylvania
ahae@cis.upenn.edu

ABSTRACT

Distributed analytics systems enable users to efficiently perform computations over large distributed data sets. Recently, systems have been proposed that can additionally protect the data's *privacy* by keeping it encrypted even in memory and by performing the computations using trusted execution environments (TEEs). This approach has the potential to make it much safer to outsource analytics jobs to an untrusted cloud platform or to distribute it across multiple parties. TEEs, however, suffer from side channels, such as timing, memory access patterns, and message sizes that weaken their privacy guarantees. Existing privacy-preserving analytics systems only address a subset of these channels, such as memory access patterns, while largely neglecting size and timing. Moreover, previous attempts to close size and timing channels suffer from high performance costs, impracticality, or a lack of rigorous privacy guarantees.

In this paper, we present an approach to mitigating timing and size side channels in analytics based on *differential privacy* that is both dramatically more efficient than the state-of-the-art while offering principled privacy assurances. We also sketch a design for a new analytics system we are developing called Hermetic that aims to be the first to mitigate the four most critical digital side channels simultaneously. Our preliminary evaluation demonstrates the potential benefits of our method.

ACM Reference Format:
Min Xu, Antonis Papadimitriou, Ariel Feldman, and Andreas Haeberlen. 2018. Using Differential Privacy to Efficiently Mitigate Side Channels in Distributed Analytics. In *EuroSec'18: 11th European Workshop on Systems Security , April 23, 2018, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3193111.3193115

1 INTRODUCTION

Recently, a number of systems have been proposed that can provide *privacy-preserving distributed analytics* [23, 29]. At a high level, these systems provide functionality that is comparable to a system like Spark [28]: users can upload large data sets, which are distributed across a potentially large number of nodes, and they can then submit queries over this data, which the system answers using a distributed query plan. However, in contrast to Spark, these systems *also* protect the confidentiality of the data. This is attractive, e.g., for cloud computing, where the owner of the data may wish to protect it against a potentially curious or compromised cloud platform.

It is possible to implement privacy-preserving analytics using cryptographic techniques [19, 21], but the resulting systems tend to have a high overhead and can only perform a very limited set of operations. An alternative approach [18, 23, 29] is to rely on *trusted execution environments (TEEs)*, such as Intel's SGX. With this approach, the data remains encrypted even in memory and is only accessible within a trusted *enclave* within the CPU. As long as the CPU itself is not compromised, this approach can offer strong protections, even if the adversary has managed to compromise the operating system on the machines that hold the data.

However, even though SGX-style hardware can prevent an adversary from observing the data itself, the adversary can still hope to learn facts *about* the data by monitoring various side channels. The classic example is a timing channel [12]. Suppose a query computes the frequency of various medical diagnoses, and suppose the adversary knows that the computation will take $51\mu s$ if Bob has cancer, and $49\mu s$ otherwise. Then, merely by observing the amount of time spent in the enclave, the adversary can learn whether Bob has cancer.

Side-channel leakage in privacy-preserving analytics has received considerable attention recently [2, 3, 17, 18, 29], but existing proposals, such as data-oblivious algorithms, mainly focus on eliminating data-dependent memory access patterns; there is much less work on addressing other side channels, such as message sizes and timing. These channels are hard to close because they fundamentally depend on the amount of data being processed and the sizes of intermediate results. As a result, the mitigations offered by prior work are unsatisfying. The most common approach is to pad computation time and message sizes all the way to their worst-case values, but this method can drive up overhead by several orders of magnitude. Furthermore, current attempts to avoid full padding either employ ad-hoc schemes that lack provable privacy guarantees [17] or rely on users to specify padding bounds a priori, which we believe is unrealistic [29].

In this paper, we sketch a new approach to mitigating timing and message size side channels in privacy-preserving analytics that is substantially more efficient than full padding, while offering

*Joint first authors with equal contributions principled privacy guarantees. We show that, if a user is willing to disclose even a *small, controlled* amount of information through side channels, they can obtain much better performance. Our main challenge is to find a principled way to reason quantitatively about this leakage, and we argue that *differential privacy* can provide such a mechanism. The information that an adversary can infer by monitoring timing and message size side channels during a query's execution is equivalent to having access to the results of a particular aggregation query over the database. Thus, just as differential privacy can be used to bound the amount of information that the query results leak about individual records, so too can it be used to bound the leakage through timing and message size side channels.

We have incorporated these insights into a prototype query planner that automatically determines the number of dummy rows that must be added to the output of query operators in order to bound side channel leakage. To do so, it computes differentially private statistics about each table involved in a query. Moreover, the planner enables users to prioritize privacy or performance and to specify how much privacy loss they are willing to tolerate for each table. It then generates an efficient query plan that respects these priorities. Our preliminary experimental evaluation shows that our approach is indeed several orders of magnitude more efficient than full padding, which is the only other principled side-channel mitigation in analytics systems so far. At the same time, it shows that our method has comparable performance to existing SGX-based analytics systems while offering stronger privacy guarantees.

The query planner is a central component of a larger system called Hermetic that we are working on. Hermetic aims to mitigate the four major digital side channels – timing, memory access patterns, instruction traces, and I/O sizes – while achieving performance that is as good or better than privacy-preserving analytics systems with weaker privacy guarantees. Hermetic is based on an *oblivious execution environment (OEE)*, a novel primitive that can perform small computations safely, by executing them in a core that is completely "locked down" and cannot be interrupted or access uncached data during the computation. In summary, our main contributions are:

- A new approach to mitigating timing and message size side channels in data analytics systems using differential privacy.
- A query planner that automatically computes the appropriate amount of padding needed to limit side channel leakage and allows users to trade off privacy and performance.
- A design for privacy-preserving analytics system that aims to be the first to mitigate the four major digital side channels.
- A preliminary experimental evaluation demonstrating the benefits of our approach.

2 BACKGROUND AND RELATED WORK

In the scenario, we are interested in, there is a group of *participants*, who each own a sensitive data set, as well as a set of *nodes* on which the sensitive data is stored. An *analyst* can submit queries that can potentially involve data from multiple nodes. Our goal is to build a distributed database that can answer these queries *efficiently* while giving strong privacy guarantees to each participant. We assume that the queries themselves are not sensitive – only their answers are – and that each node contains a trusted execution environment (TEE) that supports secure enclaves and attestation, e.g., Intel's SGX.

Threat model: We assume that some of the nodes are controlled by an adversary – for instance, a malicious participant or a third party who has compromised the nodes. The adversary has full physical access to the nodes under her control; she can run arbitrary software, make arbitrary modifications to the OS, and read or modify any data that is stored on these nodes, including the local part of the sensitive data that is being queried. We explicitly acknowledge that the analyst herself could be the adversary, so even the queries could be maliciously crafted to extract sensitive data from a participant.

2.1 Background: Differential privacy

One way to provide strong privacy in this setting is to use *differential privacy)* [8]. Differential privacy is a property of randomized queries – that is, queries do not compute a single value but rather a probability distribution over the range R of possible outputs, and the actual output is then drawn from that distribution. This can be thought of as adding a small amount of random "noise" to the output. Intuitively, a query is differentially private if a small change to the input only has a statistically negligible effect on the output distribution.

More formally, let I be the set of possible input data sets. We say that two data sets $d_1, d_2 \in I$ are similar if they differ in at most one element. A randomized query q with range R is ε-differentially private if, for all possible sets of outputs $S \subseteq R$ and all similar input data sets d_1 and d_2,

$$Pr[q(d_1) \in S] \leq e^{\varepsilon} \cdot Pr[q(d_2) \in S].$$

That is, any change to an individual element of the input data can cause at most a small multiplicative difference in the probability of *any* set of outcomes S. The parameter ε controls the strength of the privacy guarantee; smaller values result in better privacy.

Differential privacy has strong composition theorems; in particular, if two queries q_1 and q_2 are ε_1- and ε_2-differentially private, respectively, then the combination $q_1 \cdot q_2$ is $\varepsilon_1 + \varepsilon_2$-differentially private [7, 8] Because of this, it is possible to associate each data set with a "privacy budget" $\varepsilon_{\max}$ that represents the desired strength of the overall privacy guarantee, and to then keep answering queries $q_1, \ldots, q_k$ as long as $\sum_i \varepsilon_i \leq \varepsilon_{\max}$.

2.2 Strawman solution with TEEs

If side channels were not a concern, we could solve our problem roughly as follows: each node locally creates a secure enclave that contains the database runtime, and the participants use attestation to verify that the enclaves really do contain the correct code. Each participant P_i then opens a secure connection to her enclave(s) and uploads her data d_i, which is stored in encrypted form, and then sets a local privacy budget $\varepsilon_{\max,i}$ for this data. When the analyst wishes to ask a query, he must create and submit a distributed *query plan*, along with a proof that the query is ε_i-differentially private in data set d_i; the enclaves then verify whether a) they all see the same query plan, and b) there is enough privacy budget left – that is, $\varepsilon_{\max,i} \geq \varepsilon_i$ for each d_i. If both checks succeed, the enclaves execute the query plan, exchanging data via encrypted messages when necessary, and eventually add the requisite amount of noise to the final result, which they then return to the analyst. Differential privacy would ensure that a malicious analyst cannot compromise privacy, and the enclaves would ensure that compromised nodes cannot get access to data from other nodes or to intermediate results.

2.3 Related work

Side channels have been haunting TEEs for a long time. It is well known that SGX, in particular, does not (and was not intended to [11]) handle most side channels [6]. It is vulnerable to classic timing channels [12], and recent work has already exploited several others, including channels due to the sequence of memory accesses from the enclave [27], the number and size of the messages that are exchanged between the nodes [17], cache timing [5], the BTB [13], and the fact that a thread exits the enclave at a certain location in the code [26].

Many generic techniques have been proposed to mitigate microarchitectural side channels. For example, T-SGX uses transactional memory to let enclaves to detect malicious page fault monitoring [24], Raccoon rewrites programs to eliminate data-dependent branches [22], `libfixedtimefixedpoint` [1] replaces IA-32's native floating-point instructions that have data-dependent timing with constant-time versions, and recent work on oblivious algorithms [3, 18, 29] can mitigate the side channel due to memory access patterns. These techniques are effective against individual channels, but there is no obvious way to combine them; also, they tend to have a very substantial performance cost.

2.4 Approach

Unlike previous work, our strategy is based on a key observation: *leakage through timing and message size side channels is equivalent to leakage through query results.* In other words, if an adversary can observe these channels during the execution of a query, this is equivalent to giving the adversary the results of an aggregation query that computes the information that is being leaked over these channels. For example, seeing how long a selection query takes can tell the adversary how many records have been selected, and this is comparable to giving the adversary the results of a `select count` query. This insight enables us to leverage traditional database-privacy techniques – such as differential privacy – to mitigate side channels; for instance, differential privacy can tell us how much padding really needs to be added to achieve a given level of privacy, and this amount will typically be far lower than the worst-case amount. Moreover, differential privacy offers a principled way to reason *quantitatively* about side channel leakage.

We plan to use this approach in a new privacy-preserving analytics system called Hermetic. Our goal is to get the "best of both worlds": we want to mitigate the four most critical digital side channels (timing, memory accesses, instruction sequence, and message sizes) simultaneously, while achieving performance that is as good as, and ideally better than, prior privacy-preserving analytics systems.

3 THE HERMETIC SYSTEM

The architecture that we envision for Hermetic consists of a master node that coordinates query execution and several worker nodes that execute portions of queries and potentially hold portions of the data set. Nodes may be operated by different parties and do not trust other, and Hermetic clients do not trust any of the nodes. Every node has a TEE, and the only trusted components are the

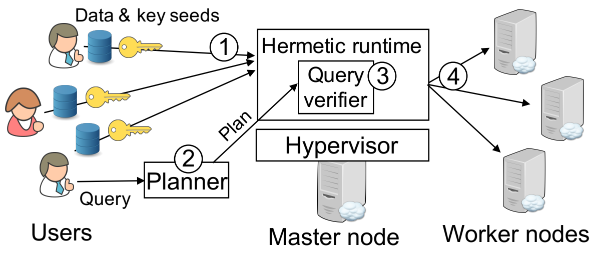


Figure 1: Hermetic's workflow.

TEEs, the software running in them, and a likely thin hypervisor (see below).

Hermetic's runtime will be comprised of the privacy-preserving query planner that is the main subject of this paper (Section 4) as well as two other components we are developing that we summarize here: OEEs for fast oblivious data processing (Section 3.1), and data-oblivious query operators (Section 3.2).

We expect Hermetic to work as shown in Figure 1: (1) The master node launches the Hermetic runtime, OEEs, and hypervisor, and clients contact the runtime to set up a master encryption key and upload their data. Attestation convinces clients that the authentic Hermetic software is running. (2) Users submit queries to the query planner, which generates a concrete query plan. (3) Since the potentially complex planner is outside Hermetic's trusted computing base, the runtime verifies the plan to make sure that every query operators is annotated with its correct sensitivity and privacy cost. (4) The master node farms out execution of individual operators from the query plan to the worker node. The workers run the run operators using a combination of OEEs and oblivious operators.

3.1 Oblivious execution environments

We are exploring a method of mitigating timing and memory-access pattern side channels for functions with small working sets. It involves preloading all their code and inputs into an isolated portion of the cache, and then running them in a "locked down" core that cannot be interrupted or monitored until the function completes. We call this primitive an *oblivious execution environment (OEE)*. More formally, an OEE executes a function $out := f(in)$, such as a merge sort, on a block of data in of limited size, while preventing an adversary from learning anything other than f and the sizes $|in|$ and $|out|$ – *even if* the adversary has access to the four digital side channels that we listed in Section 2.4.

We believe that the following four properties will be necessary and sufficient to implement OEEs:

i *OEEs preload and flush.* Before the execution on a block of data in OEEs, all the memory related to the data and the instructions have to be preloaded to OEEs. After the execution finishes, the same set of memory must be flushed from OEEs.
ii *Data-independent runtime instruction trace.* An OEEs program has to retire the identical trace of instructions, in terms of op-code, on any input the same size. Furthermore, constant-time instruction have to be enforced, as in [1].
iii *Non-preemption.* The execution, from the beginning to the end, on a block of data in OEEs should not be interrupted.

iv *Cache isolation.* All the cache lines associated with the data and code of OEEs' execution must be isolated from other concurrent processes.

It is unclear whether current CPUs support "locking down" a core to prevent preemption. If they do not, our prototype may need to implement that functionality with a thin hypervisor.

3.2 Oblivious operators

OEEs could provide a way to execute simple computations on fixed-size blocks of data, while mitigating side channels. But to answer complex queries, Hermetic will also need higher-level operators.

Our starting point is the oblivious relational operators from prior work, such as `selection`, `group-by`, and `join` [3, 18]. These operators' memory access patterns do not depend on the contents of their inputs, but they do not address timing or message size side channels. As a result, we plan to enhance these operators so that their timing and output sizes can be padded with *dummy rows* whose quantity is dictated by differential privacy. The number of dummy rows must be drawn from a Laplace distribution with parameter $\lambda = s/\varepsilon$, where the sensitivity s is determined by the query planner. Since the amount of padding is itself sensitive, this drawing must be performed in an enclave.

Current oblivious operators have two other limitations that we plan to address. First, they are often quite slow due to their reliance on data-independent sorting networks [4]. We plan to try to speed them up by breaking up the sorting task into smaller blocks, and safely sorting them using traditional merge sort in OEEs. Second, these operators still often have data-dependent instruction traces, which will likely address by eliminating conditional branches using techniques similar to Rane et al. [22].

4 PRIVACY-AWARE QUERY PLANNING

Hermetic's query planner will assemble operators like those described above to answer SQL-style queries. Query planning is a well-studied problem in databases, but Hermetic's use of differential privacy adds a twist: Hermetic is free to choose the amount of privacy budget ε it spends on each operator. Thus, it is able to trade off privacy and performance: smaller values of ε result in stronger privacy guarantees but also add more dummy rows, which slows down subsequent operators.

4.1 Computing operator sensitivities

For any query plan it considers, the query planner must first derive upper bounds on the sensitivities s_i of the plan's operators O_i. Hermetic can do so by deriving sub-queries that compute the number of rows in each operator's output [15]; we call these queries *leakage queries*. Since results of leakage queries are revealed to the untrusted query planner, they themselves must be noised with differential privacy. For example, in Figure 2, to bound the sensitivity of the leftmost join, Hermetic performs a leakage query that counts the number of rows in C where age $\leq$ 27 and noises the result. If the leakage query contains joins, s_i also depends on the multiplicities of the joined attributes, which can can also be determined in a privacy-preserving way. If each operator adds a number of dummy rows that is drawn from $Lap(s_i/\varepsilon_i)$, the overall query plan is $(\sum_i \varepsilon_i)$-differentially private.

The quantity of dummy rows drawn from $Lap(s_i/\epsilon_i)$ could be negative, however. In that case, the result would be truncated,

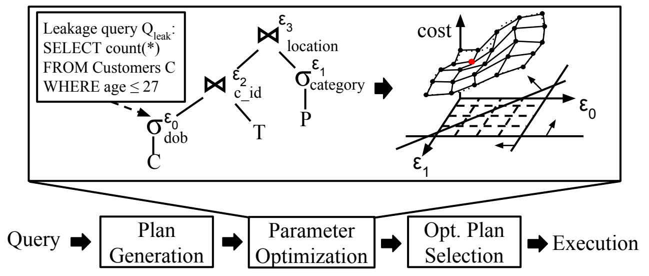


Figure 2: Query optimization for an example query. For simplicity, only the ε_0 and ε_1 dimensions of the cost space are shown.

thereby introducing some inaccuracy.[1] Nevertheless, in exchange for extra padding, we can arbitrarily reduce the probability P_{trunc} that this condition will occur by adding an offset o_i to the value sampled from the Laplace distribution. For some applications, a $P_{trunc} = 0.001$ may be acceptable.

4.2 Cost estimation

To trade off privacy and performance, the query planner must not only compute the privacy cost, which is equal to $\sum_i \varepsilon_i$, it must also estimate the performance cost. To obtain an accurate performance estimate, we model each of Hermetic's operators as a function of their input size. This can be done using an established histogram-based approach from the database literature [20]. According to this approach, the output size of selections is $N_R \cdot sel(R.a = X)$ and of joins is $N_{R_1} \cdot N_{R_2} \cdot sel(R_1.a \bowtie R_2.b)$, where N_{R_i} is the size of input relation R_i, and $sel(R.a = X)$ and $sel(R_1.a \bowtie R_2.b)$ correspond to the estimated selectivities of selection and join, respectively. As shown in [9], the selectivities can be estimated from simple statistics that Hermetic computes using the `histogram` operator. To assess the performance implications of the dummy rows, Hermetic takes them into account when estimating relation sizes. Since $Lap(s_i/\varepsilon_i)$ has a mean of zero, the expected number of dummy rows added by operator O_i is simply the offset o_i. As with leakage queries, the results of histogram queries used in cost estimation must be noised with differential privacy to avoid leaking sensitive data to the untrusted query planner.

4.3 Query optimization

Hermetic's query planner uses multi-objective optimization [25] to find the optimal plan that matches the user's priorities. A plan is associated with multiple costs, including the overall performance cost and a vector of privacy costs over the sensitive input relations. The user specifies the vectors of bounds, **B**, and weights, **W**, on the privacy costs for the input relations. Each candidate plan is represented as a join tree covering all the input relations, with each noised operator assigned a privacy parameter, ε_i. The planner outputs the plan whose weighted sum of all the costs is optimal, under the user-specified constraints.

The planner first constructs the complete set of alternative plans joining the input relations. Then, for each of the candidate plans, it formalizes an optimization problem on the privacy parameters of

[1]If the total number of rows with padding added is negative, we can safely clamp it to zero because doing so is equivalent to a postprocessing step, which does not affect the privacy cost.

all the noised operators, and solves it using linear programming. Finally, the optimal plan under the user's constraints is returned.

Returning to the query example in Figure 2, let p be the plan under consideration, $\boldsymbol{\varepsilon}[i]$ be the privacy parameter on the i-th operator of the plan, and $f_p(\boldsymbol{\varepsilon})$ be the plan's overall performance cost. Then, we could solve the following optimization problem for the privacy parameters:

$$\begin{aligned} &\min \mathbf{W} \cdot \mathbf{A} \cdot \boldsymbol{\varepsilon} + f_p(\boldsymbol{\varepsilon}) \\ &s.t.\ \mathbf{A} \cdot \boldsymbol{\varepsilon} \leq \mathbf{B}, \mathbf{0} < \boldsymbol{\varepsilon} \leq \mathbf{B}. \end{aligned} \tag{1}$$

Here, the matrix $\mathbf{A}$ is the linear mapping from privacy parameters to the privacy costs on the input relations. For instance, suppose the C, T and P relations are indexed as 0, 1 and 2, and the privacy parameters on the selection on C, the selection on P, the join of C and T and the join of ($C \bowtie T$) and P are indexed as 0, 1, 2 and 3 respectively. Then, the corresponding $\mathbf{A}$ for the plan is:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{2}$$

Finding an exact solution to this optimization problem is challenging because we cannot assume that the performance cost function $f_p(\boldsymbol{\varepsilon})$ is either linear or convex. Instead, Hermetic tackles the problem by approximating the cost function with piecewise linear approximation, and solving the piecewise linear programming problem to derive the near-optimal assignment of $\boldsymbol{\varepsilon}$. This approach is consistent with existing nonlinear multidimensional optimization techniques in the optimization literature [14].

The number of partitions of the parameter space, K, affects the optimization latency and accuracy. Larger K leads to more fine-grained linear approximation of the non-linear objective, but requires more linear programmings to be solved. To amortize the optimization overheads for large K, the query planner could be extended with parametric optimization [10] to pre-compute the optimal plans for all possible W and B so that only one lookup overhead is necessary to derive the optimal plan at runtime.

5 PRELIMINARY RESULTS

To demonstrate the potential performance benefits of using differential privacy for mitigating side channels as well as the query planner's ability to trade off privacy and performance, we now present some back-of-the-envelope calculations and preliminary results from our initial experiments.

We ran a prototype of Hermetic's query planner on an Ubuntu 14.04LTS machine that had a quad-core 2.1 GHz Intel Xeon E5-2600 processor and 64GB RAM. We used a data set consisting of three relations: taxi trips, customer information, and points of interest. The `Trips` relation has 5-days-worth of records drawn from real-world NYC taxi data [16], which has previous been used to study side-channel leakage in MapReduce [17]. Since the NYC Taxi and Limousine Commission did not release data about `Customers` or points of interest (`Poi`), we synthetically generated them. To allow for records from the `Trips` relation to be joined with the other two relations, we added a synthetic customer ID column to the trips table, and we used locations from the `Trips` relation as `Poi`'s geolocations. We used `SELECT sum(tip) FROM Customer C, Trip T, Poi P WHERE C.c_id=T.c_id AND T.drp = P.loc AND C.dob>01/01/1986 AND P.category = 'hospital'`, illustrated in Figure 2, as our example query Q_1.

Operator	Actual size	Max size	Noised size ($\epsilon = 0.01$)	Noised size ($\epsilon = 0.001$)
σ_{dob}	$400 \cdot 10^3$	$2 \cdot 10^6$	$400.92 \cdot 10^3$	$409.2 \cdot 10^3$
$\sigma_{category}$	500	$11 \cdot 10^3$	1420	$9.7 \cdot 10^3$
$\bowtie_{c\_id}$	$1.5 \cdot 10^6$	$148 \cdot 10^{12}$	$1.592 \cdot 10^6$	$2.42 \cdot 10^6$
$\bowtie_{location}$	$100 \cdot 10^3$	$1628 \cdot 10^{15}$	$4.7 \cdot 10^6$	$4.61 \cdot 10^7$

Table 1: The actual, maximum, and noised output sizes (in rows) of each operator in Q_1. The noised sizes are an upper bound that holds with probability 99%.

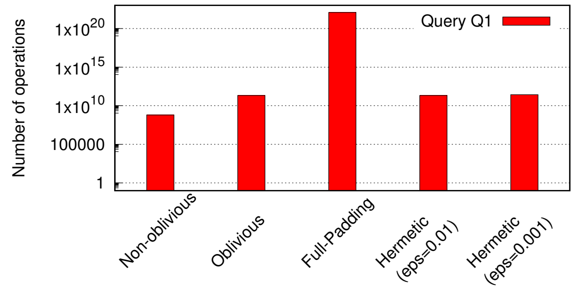


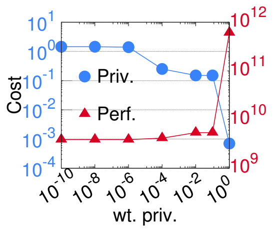
Figure 3: Estimated performance cost of different configurations running Q_1 (y-axis is logarithmic).

5.1 Benefits of differentially private padding

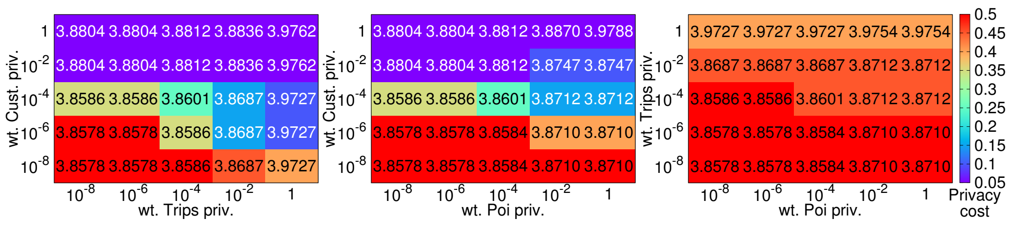
Our first goal was to estimate how well Hermetic's method of computing padding sizes with differential privacy performs compared with alternative approaches. We tested five configurations with Q_1: `Non-oblivious`, which did nothing to mitigate side channels; `Oblivious`, which used the oblivious relational algorithms from prior work [3], but did not handle timing or message size side channels, `Full-padding`, which naively mitigated side channels by padding each operator's output size up to its maximum value; and two configurations of Hermetic using different values for the privacy cost ϵ. `Non-oblivious` used the standard non-oblivious sort-merge algorithm for joins, whereas the other configurations used the oblivious join algorithm from Arasu et al. [3] which does 6 data-oblivious sorts.

Our `Customer` relation had 2 million records, our `Trip` relation had 74 million rows, and our `Poi` relation had 11 thousand points of interest. Given those relation sizes, Table 1 shows the actual, maximum, and noised size of the output of each of Q_1's relational operators. The maximum size of selections is equal to the size of the input relation, and the maximum size of a join is equal to the product of the sizes of the two input relations. The noised sizes reflect size of each operator's output assuming it was padded with noise drawn from the Laplace distribution with the given value for ϵ.

Since the performance of each operator in Q_1 depends on its input size, we were able to estimate each operator's performance using the output size of the preceding operator in the query plan. By summing these estimates, we were able to derive an estimate of the total performance cost of each of the five configurations we tested. The results, shown in Figure 3, demonstrate that, as expected, `Non-oblivious` is the fastest, but of course it offers no defense against side channels. On the other hand, `Full-padding`, previously



(a) Total privacy and perf. costs



(b) `Customers` privacy cost and overall performance cost ($\times 10^9$)

Figure 4: Privacy and performance costs for various weights on the privacy of `Customers`, `Trips` and `Poi`.

the only principled approach to mitigating timing and size channels, has huge overhead – about 13 orders of magnitude! By contrast, Hermetic's performance is much closer to `Non-oblivious` than to `Full-padding` and is roughly on par with `Oblivious`, which unlike Hermetic only mitigates memory access pattern leakage rather than timing and size channels as well.

5.2 Benefits of query optimization

Next, we explored how the query planner could derive different plans depending on the relative importance that the analyst assigned to performance as opposed to privacy and depending on the sensitivity that the analyst assigned to the data in each relation. We did so by supplying the planner different weight vectors with weights corresponding to the importance of performance and to the importance of the privacy of each relation. First, we set the privacy weights on each relation to be equal and progressively increased their relative weight with respect to performance cost. As shown in Figure 4(a), as the weight on privacy was increased, the planner found plans with different performance-privacy tradeoffs.

Second, we adjusted allowable privacy cost to the relations relative to each other. Figure 4(b) shows the privacy cost on the `Customers` relation as a function of the weights on two of the relations (a different pair in each sub-figure). In each case, the weight on the third relation was fixed to 10^{-8}. Not surprisingly, the privacy cost on `Customers` follows the weight on `Customers`, and is independent on the weights on `Trips` and `Poi`. The numbers in the cells of the heat maps indicate the performance costs of the optimal plans in each case, and they increased as the weight on any relation increased.

6 CONCLUSION

In this paper, we have presented an approach to mitigating timing and size side channels in distributed analytics systems based on differential privacy. It is much more efficient than full padding and offers more rigorous privacy guarantees than prior ad hoc methods. Our query planner that implements this method also allows analysts to trade off privacy and performance automatically. We are working on Hermetic, a new analytics system that aims to combine this query planner with oblivious execution environments and improved oblivious operators in order to to mitigate the four most critical digital side channels simultaneously. Our preliminary results suggest that Hermetic can be competitive with previous privacy-preserving systems even though it provides stronger privacy guarantees.

ACKNOWLEDGEMENTS

We thank the reviewers for their constructive feedback. This work was supported by NSF grant CNS-1513694 and UChicago CERES.

REFERENCES

[1] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *Proc. S&P*, 2015.
[2] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *Proc. CIDR*, 2013.
[3] R. K. Arvind Arasu. Oblivious query processing. In *Proc. ICDT*, 2014.
[4] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS*, 1968.
[5] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proc. USENIX WOOT*, 2017.
[6] V. Costan and S. Devadas. Intel sgx explained. Technical Report 2016/086, Cryptology ePrint Archive.
[7] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. 2006.
[8] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, 2006.
[9] B. Harangsri. *Query result size estimation techniques in database systems.* PhD thesis, The University of New South Wales, 1998.
[10] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *Proc. VLDB*, 1992.
[11] S. Johnson. Intel SGX and side channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, Mar. 2017.
[12] B. W. Lampson. A note on the confinement problem. *CACM*, 16:613–615, 1973.
[13] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proc. Security)*, 2017.
[14] R. Misener and C. A. Floudas. Piecewise-linear approximations of multidimensional functions. *J. Optim. Theory Appl.*, 145(1):120–147, 2010.
[15] A. Narayan and A. Haeberlen. Djoin: Differentially private join queries over distributed databases. In *Proc. OSDI*, 2012.
[16] NYC Taxi & Limousine Commission. TLC Trip Record Data (April, 2017). http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[17] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *Proc. CCS*, 2015.
[18] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *Proc. USENIX Security*, 2016.
[19] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *Proc. OSDI*, 2016.
[20] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Record*, 14(2):256–276, 1984.
[21] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. SOSP*, 2011.
[22] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proc. USENIX Security*, 2015.
[23] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. S& P*, 2015.
[24] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proc. NDSS*, 2017.
[25] I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In *Proc. SIGMOD*, 2014.
[26] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *Proc. ESORICS*, 2016.
[27] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. S&P*, 2015.
[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 2012.
[29] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proc. NSDI*, 2017.