HERMETIC: PRIVACY-PRESERVING DISTRIBUTED ANALYTICS WITHOUT
(MOST) SIDE CHANNELS

A THESIS SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
MIN XU

CHICAGO, ILLINOIS
JUNE, 2017
Distributed analytics systems, such as Spark, enable users to efficiently perform computations over large distributed data sets. Recently, a number of systems have been proposed that can additionally protect the privacy of the data, by keeping it encrypted even in memory, and by performing the computations using trusted hardware features, such as Intel’s SGX. This approach is attractive because it makes it much safer to outsource computations, e.g., to an untrusted cloud platform. However, existing solutions remain vulnerable to a variety of side channels, such as timing, message sizes, and cache contents, which considerably weakens the privacy guarantees that these systems can effectively provide.

In this paper, we present a principled approach to closing or mitigating the most critical side channels. We introduce a new primitive that can perform simple computations on a “locked-down core”, without externally visible side effects, as well as several enhanced oblivious algorithms that can use this primitive to answer more complex queries over large data sets. To preserve efficiency and to enable optimizations, we allow our algorithms to carefully disclose a small amount of information via differentially private queries. We present the design of a system called Hermetic that uses our techniques to answer SQL-style queries; our experimental evaluation of a Hermetic prototype shows that it is competitive with previous privacy-preserving systems, even though it provides stronger privacy guarantees.
ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Ariel Feldman, for his supports on this work. In addition to the numerous inspirational ideas that he proposed during our regular meetings, his encouragement during the down period of the project, as well as his assistance even during his own medical recovery, without which this project could by no means be completed in time, are extremely significant to me.

I also want to thank our collaborators, Antonis Papadimitriou and Professor Andreas Haeberlen, for their valuable efforts and dedications to this work. I really enjoy working with them.

I want to thank Professor Shan Lu and Professor Hank Hoffman for their suggestions on this project. I am also grateful to Bob Bartlett, Phil Kauffman and Colin Hudler for their technical helps.

Last but not least, I want to thank my parents for their encouragement and support since I was born. And I want to thank my significant other, Zixiaojie Yang, for her supports since the beginning of my PhD study.
To my parents, Linjun Xu and Guanlan Hu
# Table of Contents

ABSTRACT ......................................................... iii
ACKNOWLEDGMENTS .............................................. i
LIST OF FIGURES .................................................. v
LIST OF TABLES .................................................... vi
1 INTRODUCTION .................................................. 1
2 OVERVIEW ......................................................... 4
  2.1 Threat model ............................................... 4
  2.2 Background: Differential privacy .......................... 4
  2.3 Strawman solution with TEEs ................................. 6
  2.4 Problem: Side channels .................................... 6
  2.5 State of the art .............................................. 7
  2.6 Approach ................................................... 8
3 OBLIVIOUS EXECUTION ENVIRONMENTS ....................... 10
  3.1 What is oblivious execution? ................................. 10
  3.2 Oblivious execution environments .......................... 11
  3.3 Challenges in building an OEE today ......................... 11
  3.4 The Hermetic hypervisor ................................... 12
4 OBLIVIOUS OPERATORS ......................................... 14
  4.1 Background ................................................ 14
  4.2 Challenges ............................................... 15
  4.3 Dummy rows ............................................... 16
  4.4 Differential privacy ....................................... 17
  4.5 New primitives ............................................ 17
5 PRIVACY-AWARE QUERY PLANNING ............................. 19
  5.1 Computing operator sensitivities ............................ 19
  5.2 Cost estimation ........................................... 20
  5.3 Query optimization ....................................... 21
6 THE HERMETIC SYSTEM ........................................ 22
  6.1 Overview ................................................ 22
  6.2 Initialization ............................................. 23
  6.3 Attestation ............................................... 24
  6.4 Query submission and verification .......................... 24
  6.5 Query execution ......................................... 25
## List of Figures

2.1 Example scenario. Analyst Alice queries sensitive data that is distributed across multiple machines, which are potentially owned by multiple participants. An adversary has complete control over some of the nodes, except the CPU. 5

5.1 Different plans for query "SELECT count(*) FROM C, T, P WHERE C.cid=T.cid AND T.location = P.location AND C.age ≤ 27 AND P.category='hospital'". 20

6.1 System overview of Hermetic. 23

8.1 Instruction and memory traces of oblivious-join w/o and w/ cmov from Pin tool. 31

8.2 Cycle-resolution measurements of the actual timing of merge-sort and linear-merge inside the OEE, and their padded timing, respectively. 32

8.3 Latency of merge-sort (MS), linear-merge (LM) and hybrid-sort (HS) with increasing un-observable memory size, compared to the batcher-sort (BS). 34

8.4 Performance of select ($S_1$), groupby ($S_2$) and join ($S_3$) for different data sizes and join attribute multiplicities. 35

8.5 (a) The performance of all experimental configurations for queries $Q_4$-$Q_6$. (b) The performance of Hermetic II for $Q_4$-$Q_6$, but for plans with different privacy cost $\varepsilon$. 36
List of Tables

5.1 Performance model for the main relational operators in Hermetic. . . . . . . . . 21

7.1 L1 hit and miss latencies for merge-sort, as reported by Intel’s specifications 
\((l_{L1}, l_{L3})\), and as measured on different datasets \((l^*_L, l^*_L)\). The last columns
show the values we used in our model. All values are in cycles. . . . . . . . . . . 27

8.1 The schema and statistics of the relations used for our end-to-end experiments. 
The synthetic relation was generated for a variety of rows and multiplicities. . . 30

8.2 Experimental configurations and their resilience to different side channels. MS stands for merge-sort, BS for batcher-sort, HS for hybrid-sort, CP for cartesian product, and SMJ for sort-merge join. * denotes that the primitive was modified as described in Section 4. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
CHAPTER 1
INTRODUCTION

Recently, a number of systems have been proposed that can provide privacy-preserving distributed analytics [51, 62]. At a high level, these systems provide functionality that is comparable to a system like Spark [60]: 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 [49, 46], but the resulting systems tend to have a high overhead and can only perform a very limited set of operations. An alternative approach – which was recently applied in Opaque [62] – is to rely on trusted hardware, 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 very 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 [31]: suppose a query computes the frequency of various medical diagnoses, and 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 that is spent in the enclave, the adversary can learn whether or not Bob has cancer. Other common side channels that have been exploited in prior work include the sequence of memory accesses from the enclave [59], the number and size of the messages that are
exchanged between the nodes [43], the contents of the cache [11], and the fact that a thread exits the enclave at a certain location in the code [32, 58].

Today, system designers have basically two options for dealing with side channels, and neither of them is particularly attractive. The first option is to simply exclude some or all of these channels from the threat model: for instance, Opaque [62] explicitly declares timing channels to be out of scope. This is not very satisfying: while timing channels intuitively “do not leak very much”, prior work shows that they can in fact leak quite a bit, such as entire cryptographic keys [61]. The second option is to plug the channels by enforcing complete determinism, e.g., by using oblivious algorithms [4] and by padding computation time and message size all the way to their worst-case values. This approach is safe but even less satisfying: as we will show experimentally, full padding can drive up the overhead by several orders of magnitude.

In this paper, we propose a more principled approach, which consists of three parts. The first is a 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. This primitive, which we call an oblivious execution environment (OEE), protects against most realistic side channels and yet improves efficiency, since there is no need to use oblivious algorithms while the core is locked. The second element is a set of enhanced oblivious operators that plug not only the memory access channel – like traditional oblivious operators – but also plug or limit the three other side channels we discussed above. The third element is a query planner that combines several of these smaller computations to answer larger queries. To avoid high overheads, our query planner is allowed to release some information about the private data, but only in a carefully controlled fashion, using differential privacy [16]. By combining these three elements, it is possible to answer queries efficiently, while at the same time giving strong privacy guarantees.

We have implemented our approach in a system we call Hermetic. Since the current
SGX hardware is not yet able to fully support the “lockdown” primitive we propose, we have implemented the necessary functionality in a small hypervisor that can be run on commodity machines today. Our results from a detailed experimental evaluation of Hermetic show that our approach is indeed several orders of magnitude more efficient than full padding (which is currently the only approach that can reliably prevent side channels). The overheads are comparable to those of existing SGX-based distributed analytics systems.

We note that Hermetic is not a panacea: like all systems that are based on trusted hardware, it assumes that the root of trust (in the case of SGX, Intel) is not compromised. Also, there are physical side channels that even Hermetic cannot plug: for instance, an adversary could use power analysis [29] or electromagnetic emanations [30], or simply depackage the CPU and attack it with physical probes [53]. However, these attacks are much harder to carry out, require specialized equipment, and may be impossible to eliminate without extensive hardware changes. Our contributions are as follows:

- The OEE primitive, which performs simple computations privately, using a locked-down core (Section 3);
- A new set of oblivious operators that prevent or limit four different side channels (Section 4);
- A novel privacy-aware query planner (Section 5);
- The design of the Hermetic system (Section 6);
- A prototype implementation of Hermetic (Section 7); and
- A detailed experimental evaluation (Section 8)

In this project, I am responsible for the design and implementation of the OEE primitives and the new set of oblivious operators. I also carried out most works of the experimental evaluation. The design and implementation of the privacy-aware query planner are done by my collaborator, Antonis Papadimitriou.
CHAPTER 2
OVERVIEW

Figure 2.1 illustrates 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.

Note that this scenario is a generalization of the scenario in some of the earlier work [51, 62], which assumes that there is only one participant, who outsources a data set to a set of nodes, e.g., in the cloud.

2.1 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.

2.2 Background: Differential privacy

One way to provide strong privacy in this setting is to use differential privacy [16], one of the strongest known privacy guarantees. Differential privacy is a property of randomized queries;
Figure 2.1: Example scenario. Analyst Alice queries sensitive data that is distributed across multiple machines, which are potentially owned by multiple participants. An adversary has complete control over some of the nodes, except the CPU.

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 $\varepsilon$-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 $\varepsilon$ controls the strength of the privacy guarantee; smaller values result in better privacy. For more information on how to choose $\varepsilon$, see, e.g., [24].

Differential privacy has strong composition theorems; in particular, if two queries $q_1$ and $q_2$ are $\varepsilon_1$- and $\varepsilon_2$-differentially private, respectively, then the combination $q_1 \cdot q_2$ is $\varepsilon_1 + \varepsilon_2$-differentially private. Because of this, it is possible to associate each data set with a “privacy budget” $\varepsilon_{\text{max}}$ that represents the desired strength of the overall privacy guarantee, and to
then keep answering queries \( q_1, \ldots, q_k \) as long as \( \sum_i \varepsilon_i \leq \varepsilon_{\text{max}} \). (Note that it does not matter what specifically the queries are asking.)

2.3 Strawman solution with TEEs

At this point, it may appear that the problem we motivated above could be solved 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 \( \varepsilon_{\text{max},i} \) 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 \( \varepsilon_i \)-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, \( \varepsilon_{\text{max},i} \geq \varepsilon_i \) 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.

This approach would seem to meet our requirements: differential privacy ensures that a malicious analyst cannot compromise privacy, and the enclaves ensure that compromised nodes cannot get access to data from other nodes or to intermediate results.

2.4 Problem: Side channels

However, the strawman solution implicitly assumes that the adversary can learn nothing at all from the encrypted data or from externally observing the execution in the enclave. In practice, there are several side channels that remain observable; the most easily exploitable are:

- **Timing channel (TC)** [31]: The adversary can measure how long the computation
in the enclave takes, e.g., by reading a cycle-level timestamp counter in the CPU before entry and after exit;

- **Memory channel (MC):** The adversary can observe the locations in memory that the enclave reads or writes (even though the data itself is encrypted!), as well as their timing, e.g., by inspecting the page tables or by measuring the contents of the cache; and

- **Instruction channel (IC):** The adversary can see the sequence of instructions that are being executed, e.g., by looking at instruction cache misses; and

- **Object size channel (OC):** The adversary can see the size of any intermediate results that the enclave stores or exchanges with other enclaves.

At first glance, these channels may not reveal much information, but this intuition is wrong: prior work has shown that side channels can be wide enough to leak entire cryptographic keys within a relatively short amount of time [61]. To get truly robust privacy guarantees, it is necessary to close or at least mitigate these channels.

### 2.5 State of the art

Prior work has generally approached side channels in one of three ways. The first is to simply exclude side channels from the threat model, as, e.g., in [62]. This seems fine if the work primarily focuses on some other challenge, but the effective privacy guarantees remain weak until the solution is combined with some defense against side channels (such as the one we propose here). The second is to use heuristics to reduce the bandwidth of some of the channels, as, e.g., in [5, 25, 28]. This approach is stronger than the first, but it remains somewhat unsatisfying, since it is difficult to formally reason about the true strength of the guarantee.
The third, and most principled, approach is to remove the dependency between the sensitive data and the signal that the adversary can observe. For instance, oblivious algorithms [4] can close the memory channel by accessing the data in a way that depends only on the size of the data, but not on any specific values, and full padding can close the timing channel by padding the computation time to its worst-case value. These approaches work well, but they have two important drawbacks: 1) to our knowledge, all prior solutions close only some subset of our four channels, but not all four; and 2), as we will show experimentally in Section 8, these approaches can have an enormous overhead, sometimes by several orders of magnitude. We are not aware of an efficient, principled solution that can close all four of our channels simultaneously.

2.6 Approach

Our goal in this paper is to get the “best of both worlds”: we want to reliably close the four side channels from Section 2.4 while preserving reasonably better performance. Our approach is based on the following three ideas:

**Oblivious execution environments:** We provide a primitive called OEE that can perform small computations $out := f(in)$ entirely in the CPU cache, and such that both the execution time and the instruction trace depend only on $|in|$, $|out|$, and $f$, but not on the actual data. This completely avoids all four channels.

**New oblivious operators:** We present several oblivious operators that can be used to compose OEE invocations into larger query plans. In addition to avoiding the memory channel – like all oblivious algorithms – our operators also have a deterministic control flow (to avoid the instruction channel), they use only timing-stable instructions (to avoid the timing channel), and the size of their output is either constant or noised with “dummy rows” to ensure differential privacy (to avoid the object size channel). Notice that there is an efficiency-privacy tradeoff: more noise yields better privacy but costs performance.
Privacy-aware query planner: We describe a query planner that optimizes for both efficiency and privacy, by carefully choosing the amount of noise that is added by each oblivious operator.
CHAPTER 3
OBLIVIOUS EXECUTION ENVIRONMENTS

The first part of Hermetic’s strategy to mitigate side channels is hardware-assisted oblivious execution, using a primitive we call an oblivious execution environment (OEE).

3.1 What is oblivious execution?

The goal of oblivious execution is to compute a function $out := f(in)$ while preventing an adversary from learning anything other than $f$ and the sizes $|in|$ of the input and $|out|$ of the output - even if, as we have assumed, the adversary can observe the memory bus and the instruction trace, and can precisely measure the execution time.

Some solutions for oblivious execution already exist. For instance, Arasu et al. [4] use special oblivious sorting algorithms that perform a fixed set of comparisons (regardless of their inputs) and thus perform memory accesses in a completely deterministic way; also, compilers have been developed that emit code without data dependent branches [50], and thus have a perfectly deterministic instruction trace.

However, the existing approaches have three key limitations. First, they are very inefficient - largely because they must perform numerous dummy memory accesses to disguise the real ones or employ expensive oblivious RAMs [19, 55, 54]. Second, they tend to focus on one or two specific side channels; for instance, even if the memory trace is deterministic, the execution time can still vary if the code includes instructions whose execution time is data-dependent. Third, the existing solutions all implicitly assume that the adversary cannot interrupt the execution and inspect the register state of the CPU.
3.2 Oblivious execution environments

To provide a solid foundation for oblivious execution, we introduce a primitive \texttt{OEE} \((f, \text{in}, \text{out})\) that, for a small set of predefined functions \(f\), has the following three properties:

1. Once invoked, \texttt{OEE} runs to completion and cannot be interrupted or interfered with;
2. \texttt{OEE} loads \texttt{in} and \texttt{out} into the cache when it starts, and writes \texttt{out} back to memory when it terminates, but does not access main memory in between;
3. The execution time, and the sequence of instructions executed, depend only on \(f\), \(|\text{in}|\), and \(|\text{out}|\); and
4. The final state of the CPU depends only on \(f\).

A perfect implementation of this primitive would plug all four side channels in our threat model: The execution time, the sequence of instructions, and the sizes of the \texttt{in} and \texttt{out} buffers are constants, so no information can leak via the TC, IC, or OC. Also, the only memory accesses that are visible on the memory bus are the initial and final loads and stores, which access the entire buffers, so no information can leak via the MC. Finally, since the adversary cannot interrupt the algorithm, she can only observe the final state of the CPU upon termination, and that does not depend on the data.

Note, however, that \texttt{OEE} is allowed to perform data-dependent memory accesses \textit{during} its execution. Effectively, \texttt{OEE} is allowed to use a portion of the CPU cache as a private, un-observable memory for the exclusive use of \(f\). This is what enables Hermetic to provide good performance.

3.3 Challenges in building an OEE today

Some of the properties of an OEE can be achieved through static transformations: for instance, we can (and, in our implementation, do) achieve property \#3 by eliminating data-dependent branches and by padding the execution time to an upper bound via busy waiting.
We can also disable hardware features such as hyperthreading that would allow other programs to share the same core, and thus potentially glean some timing information from the OEE. Properties #2 and #4 can be achieved through careful implementation. Finally, by executing the OEE in a SGX enclave, we can ensure that the data is always encrypted while in memory.

However, today’s SGX unfortunately cannot be used to achieve property #1. By design, SGX allows the OS to interrupt an enclave’s execution at any time, as well as flush its data from the cache and remove its page table mappings [12]. Indeed, these limitations have already been exploited to learn the secret data inside enclaves [59, 11, 32].

### 3.4 The Hermetic hypervisor

To overcome these limitations, we use a small hypervisor. Before an OEE can execute, the hypervisor (1) completely “locks down” the OEE’s core by disabling all forms of preemption – including IPIs, IRQs, NMIs, and timers; (2) prevents the OS from observing the OEE’s internal state, by mapping or flushing any of its memory pages, or by accessing hardware features such as performance monitoring or hardware breakpoints; and (3) returns control to the enclave, so it can prefetch in and perform dummy writes to out and to the stack, so that both are in the cache and the cache lines of the latter are already in the Modified state. When the OEE completes, the cache is flushed, and the hypervisor re-enables preemption.

If the OEE’s core shares a last-level cache with other cores, the hypervisor must take care to prevent cache timing attacks. One way to do this would be to simply lock down these other cores as well; however, this would severely limit concurrency while an OEE is executing. Instead, we can use Intel’s Cache Allocation Technology (CAT) [40] to partition the cache between the OEE’s core and the other cores at the hardware level. In Section 7, we present further details of the hypervisor’s design and its use of the CAT.

We view the hypervisor as interim step that makes deploying Hermetic possible today.
Its functionality is constrained enough that it could be subsumed into future versions of SGX. We believe that this paper and other recent work on the impact of side channels in TEEs demonstrates the importance of adding OEE functionality to TEEs.
OEEs provide a way to safely execute simple computations on small amounts of data, without side channels. However, to answer complex queries over larger data, Hermetic also needs higher-level operators, which we describe next.

4.1 Background

Prior work [4, 44] has already developed a number of oblivious operators, which are guaranteed to access the data in a deterministic way. A simple example would be a projection operator that performs a linear scan over all the input tuples and extracts a particular column of interest. This already addresses a subset of the MC, since the data is accessed in a deterministic, data-independent order, so we use these operators as a starting point. However, note that there are other kinds of memory accesses (to code, to the stack, etc.), and that the timing of the accesses can be disclosive as well; we will discuss how we address these shortly.

Previous work includes some operators that are relatively standard; for instance, project, rename, union, and cartesian-product are similar to the operators found in any standard DBMS, and they are oblivious by nature, e.g., cartesian-product considers all different pairs of tuples from two relations in a data-independent way. However, there are some additional primitives that are needed for oblivious operation. The linchpin of oblivious query processing is an oblivious sort primitive. Classical sorting algorithms, such as Quicksort, would leak the ordering of the data elements, so this requires special algorithms, such as Batcher’s odd-even mergesort (batcher-sort) [7], that access the data in a deterministic order.

Building on oblivious sorting, one can implement a variety of low-level oblivious primitives. augment adds a new column to a relation and sets it to the value of a particular e-
pression; \texttt{grsum} (group-running sum) adds up the values in a column for fields that share the same key; \texttt{filter} discards all tuples that do not satisfy a predicate; \texttt{semijoin-aggregation} counts the occurrences of one relation’s attribute in another relation; \texttt{expand} creates \(k\) clones of each tuple, where \(k\) is a value that is taken from a special column; and \texttt{stitch} linearly “stitches together” two relations of the same size by combining their columns.

Higher-level operators can be implemented by combining the above primitives. For instance, to implement a \texttt{join}, one first uses \texttt{semijoin-aggregation} on each relation to compute how many matches each tuple has in the other relation, then performs an \texttt{expand} to create as many clones of each tuple, and applies \texttt{stitch} to create the result. Similarly, a \texttt{groupby} would first \texttt{sort} the relation by the group key, then apply \texttt{grsum} to add up the values in the column that is being aggregated, and finally use \texttt{filter} to leave only the last tuple for each key, which contains the total sum. These primitives, as well as the algorithms for building larger query plans from these smaller primitives, are fairly standard, so we do not discuss them in detail; instead, we focus on the points where Hermetic differs from prior work.

### 4.2 Challenges

There are three key reasons why the existing operators are not sufficient for our purposes. The first and simplest one is that prior work tends to specify the operators in pseudocode, whereas the TC and IC very much depend on the finer details of the implementation. For instance, some \texttt{x86} instructions, such as floating-point instructions and integer division, have data-dependent timing, and must be avoided to prevent the TC; similarly, to prevent the IC, we must avoid data-dependent branches (e.g., by using the \texttt{cmov} instruction) and preload the code and the stack. The necessary steps are known, and we do not claim them as a contribution.

The second reason is that some prior work [4] often assumes a client-server model: the
data is stored on the server and queries are processed on the client, which can issue read and write requests to the server. The threat model typically limits the adversary to observing the sequence of reads and writes, which excludes the TC and IC entirely and limits the MC to only data accesses. In other words, accesses to code or the stack are not considered, and it is assumed that the adversary cannot observe the precise timing of the accesses. In Hermetic, we make use of the OEE to block these channels; however, since the OEE is limited to small data sets, we also need operators to handle larger workloads.

The third reason is that, to our knowledge, prior work pays almost no attention to the OC: a select either reveals the exact number of rows that match the predicate, which is disclosive, or massively pads the output to the worst-case size, which is inefficient. To address this, we introduce a new technique, which we discuss next.

4.3 Dummy rows

To address the OC, Hermetic can pad relations with dummy tuples, so they appear larger (to the adversary) than the actual data tuples they contain. In essence, we add a special isDummy column to each relation, which is set to 1 for dummy tuples and to 0 otherwise, and we augment all the oblivious operators to ignore tuples that have isDummy set to 1. The latter is necessary to maintain correctness: for instance, we must prevent the dummy tuples from appearing in the output of a select, and we achieve this by adding an & & !isDummy to each predicate in the filter.

Hermetic also needs a way to introduce dummy tuples in all the operators that have a variable-size output (i.e., whose output size is not a deterministic function of their input size) – specifically, select, groupby, and join. We modified the above three operators to take an extra parameter that specifies the number of dummy tuples to add. For details please refer to Appendix B.
4.4 Differential privacy

To ensure differential privacy, the number of dummy tuples that a given operator must add needs to be drawn from a Laplace distribution with parameter $\lambda = s/\varepsilon$, where $\varepsilon$ is the privacy parameter (Section 2.2) and $s$ is the sensitivity of the output size to the input data – the maximum change in the output size that can result from adding or removing one input tuple. The sensitivity can be determined by the query planner (Section 5), but the actual Laplace value has to be drawn in the enclave, when the query is being executed. It is critical that this draw be immune to our four side channels as well: if the adversary can learn the value that is being drawn, she can compute the actual number of data tuples in the output and thus potentially learn facts about the input data.

To guard against this, we developed a special operator that can draw values from a Laplace distribution in constant time, with a deterministic control flow and without accessing main memory. To some degree, this involves only careful programming, e.g., to ensure that only static loop bounds are used; however, one important challenge is that the floating-point instructions on x86 cannot be used because they have data-dependent timing [2]. We instead used a fixed-point math library based on [2], which uses integer instructions.

4.5 New primitives

Finally, Hermetic needs three additional oblivious primitives. The first two are histogram, which computes a histogram over the values in a given column, and multiplicity, which computes the multiplicity of a column – that is, the number of times that the most common value appears. These operators are used by the query planner to compute statistics about the input data that it needs to find the best privacy-efficiency tradeoff. We discuss these more in Section 5.

The third new primitive is hybrid-sort, which can obliviously sort large relations by
repeatedly invoking oee to sort smaller chunks of data using a fast, non-oblivious sorting algorithm (mergesort), and to combine the results by merging chunks of data using a non-oblivious merging algorithm (linear-merge). hybrid-sort is essentially a block-based variant of batcher-sort that takes advantage of the OEE, so the obliviousness properties of batcher-sort trivially apply to hybrid-sort as well.
CHAPTER 5
PRIVACY-AWARE QUERY PLANNING

Next, we describe how Hermetic assembles the operators from Section 4 into query plans that can compute the answer to SQL-style queries. Query planning is a well-studied problem in databases, but Hermetic’s use of differential privacy adds an interesting twist: Hermetic is free to choose the amount of privacy budget \( \varepsilon \) it spends on each noised operation. Thus, it is able to make a tradeoff between privacy and performance: smaller values of \( \varepsilon \) result in stronger privacy guarantees but also add more dummy tuples, which slows down the downstream operators.

5.1 Computing operator sensitivities

For any query plan it considers, Hermetic must first derive upper bounds on the sensitivities of all the operators \( O_i \) in the plan. To do this, Hermetic derives sub-queries that compute the number of tuples in each operator’s output; we call these query the leakage queries of the operators. Figure 5.1 illustrates how this is done for a simple example query (shown in the caption): the number of tuples that are output by the selection operator in Figure 5.1(a) is simply the number of customers who are at most 27 years old.

Once an operator’s leakage query is known, Hermetic applies an algorithm from [38] to compute an upper bound on its sensitivity \( s_i \). If the leakage query contains joins, the algorithm needs to know the multiplicities of the joined attributes; Hermetic obtains these using the multiplicity operator from Section 4.5. Once \( s_i \) is known, Hermetic annotates the operator accordingly. If each operator adds a number of dummy tuples that is drawn from \( \text{Lap}(s_i/\varepsilon_i) \), the overall query plan is \((\sum_i \varepsilon_i)\)-differentially private.

However, drawing from \( \text{Lap}(s_i/\varepsilon_i) \) can return negative values, but the padding must be positive to avoid deleting useful results. Hence, Hermetic actually adds \( o_i + \text{Lap}(s_i/\varepsilon_i) \)
dummy tuples, where \( o_i \) is a tunable parameter. If the value drawn from \( \text{Lap}(s_i/\varepsilon_i) \) at runtime is smaller than \(-o_i\), the query fails and has to be retried. Notice that this means that Hermetic technically provides \((\varepsilon, \delta)\)-differential privacy [15], a standard generalization of differential privacy.

5.2 Cost estimation

As discussed earlier, choosing the \( \varepsilon_i \) values in a query plan involves a tradeoff between privacy and performance. The privacy cost of a query plan is simply \( \sum_i \varepsilon_i \), so we focus on estimating the performance. To obtain a performance model, we derived the complexity of the Hermetic operators as a function of their input size; the key results are shown in Table 5.1.

To estimate the size of intermediate results, we used an established histogram-based approach from the database literature [48]. According to this approach, the output size of selections is \( N_R \cdot \text{sel}(R.a = X) \) and of joins is \( N_{R_1} \cdot N_{R_2} \cdot \text{sel}(R_1.a \bowtie R_2.b) \), where \( N_{R_i} \) is the size of input relation \( R_i \), and \( \text{sel}(R.a = X) \) and \( \text{sel}(R_1.a \bowtie R_2.b) \) correspond to the estimated selectivities of selection and join, respectively. As shown in [23], the selectivities can be estimated from simple statistics that Hermetic computes using the histogram operator from Section 4.5.

To enable the planner to assess the performance implications of the dummy tuples, Hermetic takes them into account when estimating relation sizes. Since \( \text{Lap}(s_i/\varepsilon_i) \) has a mean of zero, the expected number of dummy tuples added by operator \( O_i \) is simply the offset \( o_i \).
Table 5.1: Performance model for the main relational operators in Hermetic.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>HybridSort</td>
<td>( h(n) = n \cdot \log(c) + n \cdot \log^2(n/c) )</td>
</tr>
<tr>
<td>Select</td>
<td>( h(n) )</td>
</tr>
<tr>
<td>Group-by</td>
<td>( 3 \cdot h(n) )</td>
</tr>
<tr>
<td>Join</td>
<td>( 4 \cdot h(n+m) + 2 \cdot h(m) + 3 \cdot h(n) + 2 \cdot h(k) )</td>
</tr>
</tbody>
</table>

5.3 Query optimization

Hermetic’s query planner uses relational algebra rules to generate all possible plans, and then picks an optimal plan based on the estimated privacy and performance costs. Since, in the applications we consider, privacy is usually more important than performance, we designed our optimizer to select the fastest plan whose privacy cost does not exceed a budget \( \varepsilon_B \), which can be set by the analyst.

Figure 5.1 illustrates some of the decisions the query planner makes. The first decision has to do with choosing the \( \varepsilon_i \) parameter of each operator, which affects both performance and privacy cost. Plans (a) and (b) show two plans that have the same structure but use a different \( \varepsilon \) parameters; plan (a) achieves better performance because the larger \( \varepsilon_i \) implies smaller intermediate results, whereas plan (b) consumes less privacy budget because the \( \sum \varepsilon_i \) is smaller.

Another decision of the planner is illustrated by plans (b) and (c). These plans use the same \( \varepsilon \), but the order of joins is different, and this affects how much \( \varepsilon \) is consumed from the budget of each relation (e.g., plan (b) uses 0.003 from C’s budget, but (c) uses 0.002). Finally, plan (d) uses Cartesian products, which do not consume any \( \varepsilon \); thus, the plan achieves the smallest total \( \varepsilon \) consumption.

Notice that \( \varepsilon_i \) can be chosen independently for each operator, which gives rise to an interesting optimization problem. Our current implementation just tries a few standard values, but a more sophisticated optimizer could potentially find plans that are more private and/or more efficient.
Next, we describe Hermetic’s design and how it combines the techniques described in the previous sections to execute relational queries on sensitive data.

6.1 Overview

Hermetic consists of a master node, and several worker nodes. Each node runs the trusted hypervisor (Section 3), and the trusted runtime that performs the Hermetic operators (Section 4) inside a secure enclave. The last component of Hermetic is the query planner from Section 5. Figure 6.1(a) shows the system components of Hermetic. The hypervisor and the query runtime are trusted, and require careful bootstrap to establish the chain of trust, together with a proof of such chain, to convince the client that the necessary security components have been correctly loaded. The query planner is untrusted, and it generates physical execution plans that will be executed only if they pass the verifications inside the trusted runtime. Figure 6.1(b) shows the workflow of Hermetic, which consists of the following steps:

1. Initially, the master node launches the Hermetic hypervisor and runtime, and the users contact the runtime to setup a master encryption key and upload their data (Section 6.2). Users are convinced of the authenticity of both the hypervisor and the runtime by using attestation (Section 6.3).

2. After initialization, users can submit queries to the Hermetic optimizer, which generates a concrete query plan and sends it to the runtime for execution (Section 6.4).

3. Since the optimizer is outside Hermetic’s trusted code base, the runtime verifies incoming plans to make sure that all operators are annotated with the appropriate sensitivity and epsilon (Section 6.4).
4. The runtime asks Hermetic worker nodes to execute the operators of the query plan using Hermetic’s oblivious operators. Afterwards, the results are returned to the user who submitted the query (Section 6.5).

Next, we describe these steps in greater detail.

### 6.2 Initialization

Hermetic is initialized after the users setup a master encryption key and upload their sensitive data to the server. Since no party in Hermetic is completely trusted, the master key is created inside the trusted runtime using randomness contributed by the users. After that, the key is encrypted using a hardware-based key and persisted to secondary storage using, e.g., Intel SGX’s sealing infrastructure [1].

With the master key in place, users send their data to the runtime, which encrypts it with the key and stores it to the disk. Since the size of the sensitive data can reveal sensitive information too, we assume that users will pad the initial relations with noise before uploading them to the runtime. At the same time, the privacy budget of each uploaded relation is initialized and stored to the disk.

To protect from replay attacks [9, 35, 10], where an old version of the privacy budget
is reused by the system, Hermetic uses a trusted non-volatile hardware counter to add a monotonic value to the privacy budget, which is then checked to confirm the budget is fresh. Such a counter is available in trusted hardware solutions, including the current version of SGX [26]. To enable the query optimizer to calculate the privacy and performance cost of query plans, some basic statistics are also computed during initialization.

6.3 Attestation

A prerequisite for uploading sensitive data is that users can be convinced that they are sending the data to a correct instantiation of the Hermetic system. This means that they need to make sure that the Hermetic hypervisor is running on the remote machine, and that the trusted hardware runs the Hermetic runtime. We achieve this level of trust as follows. Upon launch, the Hermetic runtime uses Intel’s trusted execution technology (TXT) [27] to get an attestation of the boot process and the hypervisor loaded at the time. If the Hermetic runtime is started on a machine without the hypervisor, it halts and performs no processing. In addition to that, users leverage enclave attestation, e.g., Intel SGX attestation [1], to get a signed measurement that the correct codebase has been loaded to the trusted hardware.

6.4 Query submission and verification

Users write their queries in a subset of SQL that supports selections, projections, joins, and group-by aggregations. Users can supply arbitrary predicates, but they cannot run arbitrary user-defined functions. User submit queries to the optimizer, which is outside Hermetic’s TCB and can reside either at the client or at the server. The optimizer then prepares a query plan to be executed by the runtime.

As explained in Section 5, query plans are annotated with the sensitivity of each relational operator, as well as with the epsilon to be used to add noise to the intermediate results. Since
the optimizer is not trusted, these privacy parameters have to be verified before the plan is executed: Hermetic has to check that the sensitivities are correct, and that the total ε annotations do not exceed the privacy budgets. Sensitivities are verified by computing them from scratch based on the query plan, and comparing them against the ones attached to the incoming plan.

6.5 Query execution

If a plan is verified to be correct, it proceeds to be executed by the runtime. Before execution starts, the privacy budget is decreased based on the epsilons in the plan, and the runtime generates the Laplace noise which determines the number of fake records to pad intermediate results with. To execute a query plan, the Hermetic runtime sends all the individual operators of the plan to different Hermetic worker nodes, which in turn use the appropriate operators from Section 4 to perform the computation.
CHAPTER 7
IMPLEMENTATION

To confirm our design and measure the performance implications of our approach, we implemented a prototype of Hermetic, which we describe next.

7.1 Hermetic hypervisor

We based the Hermetic hypervisor on Trustvisor [36], a compact extensible hypervisor that has been formally verified [56]. To support the functionality needed for establishing an OEE, we extended Trustvisor with two hypercalls, named LockCore and UnlockCore, respectively. The LockCore hypercall performs the following actions: (1) it checks that hardware hyper-threading and prefetching are disabled (these can be done by checking the number of logical and physical cores using CPUID, and using techniques from [57]), (2) it disables interrupts and preemption (3) it disables the RDMSR instruction for non-privileged instructions to prevent snooping on package-level performance counters of the Hermetic core, (4) it flushes all cache lines (with WBINVD), (5) it uses CAT to assign a part of the LLC exclusively to the core running Hermetic (by writing to several model-specific registers [41]), and (6) it returns control to the Hermetic runtime that called the hypercall. After calling LockCore, the Hermetic runtime can proceed with the remaining steps required to establish an OEE (Section 7.2). UnlockCore reverts actions (2) to (5) in the reverse order. Overall, we modified 300 SLoC within the Trustvisor code.

7.2 Oblivious execution environment

Memory-access obliviousness: To ensure that all memory accesses from within the OEE are served from the cache, we disabled hardware prefetching, as discussed in Section 7.1, and we pre-loaded all data and instructions to the cache. For the former we used the prefetcht0
Table 7.1: L1 hit and miss latencies for merge-sort, as reported by Intel’s specifications ($l_{L1}$, $l_{L3}$), and as measured on different datasets ($l^*_L1$, $l^*_L3$). The last columns show the values we used in our model. All values are in cycles.

<table>
<thead>
<tr>
<th>Data</th>
<th>$l_{L1}$</th>
<th>$l_{L3}$</th>
<th>$l^*_L1$</th>
<th>$l^*_L3$</th>
<th>$l_{L1}$</th>
<th>$l_{L3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>4</td>
<td>34</td>
<td>0.68</td>
<td>3.34</td>
<td>0.74</td>
<td>5.0</td>
</tr>
<tr>
<td>Ordered</td>
<td></td>
<td></td>
<td>0.6693</td>
<td>3.8032</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse</td>
<td></td>
<td></td>
<td>0.6664</td>
<td>4.263</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

instruction, which instructs the CPU to keep the data cached, and we performed dummy writes to prevent leakage through the cache coherence protocol. For the latter, we adjusted our algorithms so that they could be run in a “shortcut mode” that exercises all the code (and thus loads it into the icache) but does not touch any actual data. We also carefully aligned all buffers in memory to avoid cache collisions.

**Timing obliviousness:** To prevent the data-dependent memory accesses inside OEE from causing variations in the execution time, we pad the execution time to a safe upper bound. In principle, this could be done by adding up the worst-case latencies of all instructions and the LLC hit latency $l_{L3}$ for all memory accesses, but this would be wildly conservative. To get a tighter bound, we carefully analyzed the two algorithms (mergesort and linear-merge) that Hermetic needs to run inside the OEE; fortunately, because of the way these algorithms access the data, it is easy to prove for many memory accesses that they will be served from the L1 cache, which allows us to substitute the L1 hit latency $l_{L1}$ for $l_{L3}$ in these cases. However, due to the superscalar execution in modern CPUs, the resulting bound is still 10x larger than the actual execution time. To further improve the bound, we performed a large number of experiments in which we measured the L1 hit and miss rates using the CPU’s performance counters, and we used regression to learn effective L1 and L3 hit latencies $l^*_L1$ and $l^*_L3$. Table 7.1 shows the effective latencies we estimated for mergesort, as well as those of the specification. Since we could not be sure that we have observed the worst case in our experiments, we added generous safety margins to obtain bounds $\hat{l}_{L1}$ and $\hat{l}_{L3}$, which our prototype uses to determine how much to pad the execution time. The resulting times are
roughly twice the actual execution times, and they were never exceeded in our experiments. For extra security, the bounds from the specification could be used instead, at the expense of somewhat lower performance. For more details please refer to Appendix A.

Hermetic’s trusted codebase consists of the runtime, which had 3,995 SLoC in our prototype, and the trusted hypervisor, which had 14,095 SLoC. The former seems small enough for formal verification, and the latter has, in fact, been formally verified [56] prior to our modifications. (We have not yet updated the proof, but it should not be difficult.) Notice that the hypervisor would no longer be necessary with a future revision of SGX that natively supports OEEs.
Next, we report results from our experimental evaluation of Hermetic. Our experiments try to answer the following questions: (1) Does Hermetic’s OEE satisfy the security properties outlined in Section 3? (2) What is the overhead of time padding inside the OEE? (3) How does having a OEE affect the performance of oblivious sorting? (4) What are the performance characteristics of Hermetic relational operators for data with different statistics? (5) Can Hermetic scale to realistic datasets and queries? (6) Can we get good performance even if we want very strong privacy guarantees?

8.1 Experimental setup

Since no existing CPU supports both SGX and CAT, we chose to experiment on an Intel Xeon E5-2600 v4 2.1GHz machine, which supports CAT, has 4 cores that share a 40 MB LLC, and features 64GB of RAM. This means that the numbers we report do not reflect any overheads due to encryption in SGX, but, as previous work [62] reports, the expected overhead of SGX in similar data-analytics applications is usually less than 2.4x. We installed the Hermetic hypervisor and Ubuntu (14.04LTS) with kernel 3.2.0. We disabled hardware multi-threading, turbo-boost, and HW prefetching because they can cause timing variations.

Table 8.2 shows the different system configurations we compared, and the side channels they defend against. NonOblivious corresponds to commodity systems that take no measure against side-channels; OblMem-NoOEE uses memory-oblivious operators from previous work [4, 44], without any un-observable memory; Full-Padding performs all joins by computing the Cartesian join, which produces outputs equal to the maximum possible size of joins; and Hermetic I and Hermetic II implement the techniques described in this paper – the only difference being that the former does not add noise to the intermediate results.
Table 8.1: The schema and statistics of the relations used for our end-to-end experiments. The synthetic relation was generated for a variety of rows and multiplicities.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Rows</th>
<th>Multiplicities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trips</td>
<td>$10^7$</td>
<td>$m(\text{cid})=32$, $m(\text{location})=1019$</td>
</tr>
<tr>
<td>Customers</td>
<td>$4 \cdot 10^6$</td>
<td>$m(\text{cid})=1$</td>
</tr>
<tr>
<td>Poi</td>
<td>$10^4$</td>
<td>$m(\text{location})=500$</td>
</tr>
<tr>
<td>Synthetic</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 8.2: Experimental configurations and their resilience to different side channels. MS stands for merge-sort, BS for batcher-sort, HS for hybrid-sort, CP for cartesian product, and SMJ for sort-merge join. * denotes that the primitive was modified as described in Section 4.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Sort</th>
<th>Join</th>
<th>MC</th>
<th>IC</th>
<th>TC</th>
<th>OC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NonOblivious</td>
<td>MS</td>
<td>SMJ</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OblMem-NoOEE</td>
<td>BS</td>
<td>[4, 44]</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Full-Padding</td>
<td>BS</td>
<td>CP</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Hermetic I</td>
<td>HS</td>
<td>[4, 44]*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Hermetic II</td>
<td>HS</td>
<td>[4, 44]*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 8.1 lists all the relations we used in our experiments. The Trips relation has 5-days-worth of records from a real-world dataset with NYC taxi-trip data [42]. This dataset has been previously used to study side-channel leakage in MapReduce [43]. Since the NYC Taxi and Limousine Commission did not release data about the Customers and points of interest (Poi) relations, 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. To examine the performance of Hermetic for data with a variety of statistics, we use synthetic relations with randomly generated data in all fields, except those that control the statistics in question.
8.2 OEE security properties

Our first experiment is designed to verify that our implementation of the OEE primitive really does have stable timing and does not access main memory during the computation. To test the behavior of merge-sort and linear-merge on a wide range of input data sets, we created synthetic relations, with randomly-generated values and as many rows as needed to completely fill the available cache (187,244 rows of 24 bytes each).

As an initial sanity check, we used the Pin instrumentation tool [34] to record instruction traces and memory accesses of the oblivious-join on two distinct input datasets of the same size. The differentiations of the two sets of traces are shown in Figure 8.1. Figures 8.1(a) and 8.1(c) show that, without the cmov instruction, using the oblivious algorithms alone cannot eliminate the variations in memory and instruction traces completely. Figures 8.1(b) and 8.1(d) confirm that the oblivious algorithm plus the cmov instruction could eliminate the
variations. We also used Intel’s performance counters to read the number of LLC misses\(^1\) and the number of accesses that were served by the cache\(^2\); as expected, we did not observe any LLC misses in any of our experiments. Finally, we used \texttt{objdump} to inspect the compiler-generated code for instructions with operand-dependent timing; as expected, there were none.

Next, we used the CPU’s timestamp counter to get cycle-level measurements of the execution time within the OEE, with and without padding. Figure 8.2 shows our results: as expected, the execution time without padding varied somewhat between data sets, but with padding, the difference between the maximum and minimum execution times was only 44 and 40 cycles, respectively. As Intel’s benchmarking manual [45] suggests, this residual variation can be attributed to inherent inaccuracies of the timing measurement code.

\(^1\) Using the \texttt{LONGEST\_LAT\_CACHE\_MISS} counter.

\(^2\) Using the \texttt{MEM\_UOPS\_RETIRED\_ALL\_LOADS} and \texttt{MEM\_UOPS\_RETIRED\_ALL\_STORES} counters.
8.3 Overhead of time padding

Next, we examine the overheads of padding time for `mergesort` and `linear-merge` in the OEE, and how they depend on the size of the un-observable memory.

Analogous to Section 8.2, we generated random data and created relations with enough rows to fill up a cache of 1MB to 27MB. On this data, we measured the time required to perform the actual computation of the two primitives, and the time spent busy-waiting to pad the execution time. We collected results across 10 runs and report the average in Figure 8.3(a). The overhead of time padding ranges between 34.2% and 61.3% for `merge-sort`, and between 95.0% and 97.9% for `linear-merge`. Even though the padding overhead of `merge-sort` is moderate, it is still about an order of magnitude faster than `batcher-sort`. This performance improvement over `batcher-sort` is enabled by having an OEE, and it is the main reason why Hermetic is more efficient than OblMem-NoOEE, even though Hermetic provides stronger guarantees.

8.4 Performance of hybrid-sort using OEE

Running `merge-sort` in the OEE is crucial for the good performance of `hybrid-sort`, which is the basic primitive on which all of the operators are built. To understand how these benefits depend on the size of the un-observable memory, we conducted the following experiment. We generated several synthetic relations of 2, 4, and 8 million random rows, and we measured the time required to sort them using `batcher-sort` and `hybrid-sort`. We repeated this experiment for un-observable memory sizes that ranged from 1MB to 27MB, and we report the results in Figure 8.3(b).

The results show that the larger the un-observable memory, the greater the benefits `hybrid-sort` provides. Also, as expected, the speedup compared to `batcher-sort` increases with the size of the relation. This illustrates the benefits of un-observable memory for efficient
8.5 Performance of Hermetic’s relational operators

Next, we examined the performance of Hermetic’s relational operators: select, groupby and join. For this experiment we used three simple queries \( (S_1 - S_3) \), whose query execution plans consist of a single relational operator. \( S_1 \) selects the rows of a relation that satisfy a predicate, \( S_2 \) groups a relation by a given column and counts how many records are per group. \( S_3 \) simply joins two relations. To understand the performance of the operators based on a wide range of parameters, we generated relations with different statistics (e.g., selection and join selectivities, join attribute multiplicities) and used NonOblivious, OblMem-NoOEE, and Hermetic to execute queries \( S_1 - S_3 \) on these relations.

Figure 8.4(a) shows the results for queries \( S_1 \) and \( S_2 \) for relations of different size. In terms of absolute performance, one can observe that Hermetic I can scale to relations with millions of records, and that the actual runtime is in the order of minutes. This is definitely slower than NonOblivious, but it seems to be a acceptable price to pay, at least for some
applications that handle sensitive data. In comparison to OblMem-NoOEE, Hermetic I achieves a speedup of about 2x for all data sizes. \( S_3 \) displays similar behavior for increasing database sizes.

We also examined the performance of Hermetic II for query \( S_3 \) on relations of different multiplicities. The amount of noise added to the output in order to achieve the differential privacy guarantee is proportional to \( s/\varepsilon \), and sensitivity \( s \) is equal to the maximum multiplicity of the join attribute in the two relations. This means we expect to see a point for large multiplicity and small \( \varepsilon \) where the noise become large enough to affect the performance considerably. Figure 8.4(b) shows that this threshold point is reached only for very small \( \varepsilon \) and large multiplicity (around 200). However, the overhead of padding in Hermetic II is small for most combinations of multiplicity and epsilon.

### 8.6 Performance on realistic datasets and queries

Finally, we compared the different system configurations on complex query plans, each of which consists of at least one \texttt{select}, \texttt{groupby}, and \texttt{join} operator. To perform this exper-
Figure 8.5: (a) The performance of all experimental configurations for queries $Q_4$-$Q_6$. (b) The performance of Hermetic II for $Q_4$-$Q_6$, but for plans with different privacy cost $\varepsilon$.

For the experiment, we used the relations described in Table 8.1, as well as three queries that perform realistic processing on the data. $Q_4$ groups the Customer relation by age and counts how many customers gave a tip of at most $10. Q_5$ groups the points of interest relation by category, and counts the number of trips that cost less than $15$ for each category. $Q_6$ counts the number of customers that are younger than 30 years old and made a trip to a hospital.

We measured the performance of all systems on these three queries, and the results are shown in Figure 8.4. Full-Padding was not able to finish, despite the fact that we left the queries running for 7 hours. This illustrates the huge cost of using full padding to combat the OC. In contrast, Hermetic II, which pads using differential privacy, has only a small overhead relative to non-padded execution (Hermetic I). This suggests that releasing a controlled amount of information about the sensitive data can lead to considerable savings in terms of performance. Also, note how hybrid-sort helps Hermetic be more efficient than previous oblivious processing systems (OblMem-NoOEE), even though it offers stronger guarantees. Overall, the performance results of Hermetic are consistent with results presented in...
in Opaque [62], where the authors assumed the existence of an un-observable memory, and reported an average of 22x and a maximum of 63x overhead compared to NonOblivious. Even though Hermetic actually implements a real OEE and pads the OC, our results are in the same ballpark.

8.7 Performance-privacy tradeoff

The last question we answer is whether we can get good performance even if we need very strong privacy guarantees.

To answer this, we used our optimizer from Section 5 to identify the fastest query plans for $Q_6$ given three different bounds on the privacy consumption $\varepsilon$ of the query (0.1, 0.01, and 0.004). Figure 8.5 shows the time taken by Hermetic II to perform these optimal plans. We can see that increasing the amount of noise added to the intermediate results, i.e., decreasing epsilon the epsilon consumption of $Q_6$, has only a limited effect on performance for all queries, and the total time remains better than the time required by OblMem-NoOEE. This suggests that, even when dealing with realistic queries and data, Hermetic can achieve strong privacy guarantees by adjusting $\varepsilon$ to small values (larger noise), at only a reasonable performance cost.
CHAPTER 9
RELATED WORK

Analytics on encrypted data: In principle, privacy-preserving analytics could be achieved with fully homomorphic encryption [17] or secure multiparty computation [8], but these techniques are still orders of magnitude too slow to be practical [18, 8]. As a result, many systems use less than fully homomorphic encryption that enables some queries on encrypted data but not others. This often limits the expressiveness of the queries they support [46, 47]. In addition, some of these systems [49, 37, 20] have been shown to leak considerable information [39, 14, 21].

Alternatively, several systems [3, 6, 51] rely on TEEs or other trusted hardware. As in Hermetic, sensitive data is protected with ordinary encryption, but query processing is performed on plaintext inside enclaves. But, due to the limitations of TEEs discussed earlier, these systems do not address side channels.

Oblivious data analytics: Recent work has begun to focus on these side channels, but so far existing systems only address one or two, often incompletely. M2R [13] and Ohrimenko et al. [43] aim to mitigate the OC in MapReduce. Both systems reduce OC leakage, but they use ad hoc methods that still leak information about the most frequent keys. By contrast, Hermetic’s OC mitigation, based on differential privacy, is more principled.

To address the MC in databases, Arasu et al. introduce a set of data-oblivious algorithms for relational operators, some based on sorting networks [4]. Ohrimenko et al. extend this set with oblivious machine learning algorithms [44]. Hermetic enhances these algorithms by making them resistant to IC, TC, and OC leakage and by speeding them up significantly using an OEE.

Opaque by Zheng et al. [62] is the system most similar to Hermetic. It combines TEEs, oblivious relational operators, and a query planner. Interestingly, Zheng et al. also recognize that performance gains are possible when small data-dependent computations can be per-
formed entirely in the CPU cache (i.e., an OEE). But, unlike Hermetic, they not describe how to realize an OEE. Furthermore, Opaque does not mitigate the IC or TC, and its mitigation for the OC relies on adding padding up to a bound determined a priori. Choosing this bound would be difficult for users: one that is too low risks privacy whereas one that is too high creates high overhead. By contrast, Hermetic computes noise automatically.

**Mitigating side channels:** It is well known that SGX does not handle most side channels [12], and recent work has already exploited several of them. These include side channels due to cache timing [11], the BTB [32], and page faults [59].

Many techniques have been proposed to mitigate so-called digital side channels that can be monitored by malware running on the target system. These include both new hardware designs and code transformations. For example, T-SGX uses transactional memory to let enclaves to detect malicious page fault monitoring [52]. CATalyst uses Intel’s CAT to mitigate cross-VM LLC side channels [33]. Raccoon is a compiler that rewrites programs to eliminate data-dependent branches [50]. Its techniques inspire the manual code modifications that Hermetic uses to mitigate the IC and TC. Hermetic also uses `libfixedtimefixedpoint` [2], a math library that replaces Intel’s native floating point instructions, which suffer from data-dependent timing.

Hermetic cannot fully mitigate physical side channels, such as power analysis [29] or electromagnetic emanations [30] because the underlying CPU does not guarantee that its instructions are data-independent with respect to them. However, these channels are most often exploited to infer a program’s instruction trace. Thus, by making queries’ instruction traces data-independent, Hermetic likely reduces these channels’ effectiveness.

Oblivious RAMs [19, 55, 54] can eliminate memory access pattern leakage in arbitrary programs, but they suffer from poor performance in practice [50]. Moreover, ORAMs only

---

1. In fact, Zheng et al. report performance results assuming the existence of an OEE with 8MB of memory.
2. e.g., Intel claims that AES-NI is resilient to digital side-channels, but does not mention others [22].
hide the addresses being accessed, not the number of accesses, which could itself leak information [62].
CHAPTER 10

CONCLUSION

In this paper, we have presented a principled approach to closing the four most critical side channels: memory, instructions, timing, and output size. Our approach relies on a new primitive, hardware-assisted oblivious execution environments, as well as a number of new oblivious operators and a novel privacy-aware query planner. We have presented a system called Hermetic that uses this approach; our experimental evaluation shows that Hermetic is competitive with previous privacy-preserving systems, even though it provides stronger privacy guarantees.
References


APPENDIX A
DETERMINING THE TIMING BOUND OF AN OEE

The discussion in Section 7.2 mentions two main techniques we used to derive a safe and efficient upper bound on the time required to execute `merge-sort` and `linear-merge` in the OEE: (1) analyzing the two algorithms to count how many accesses are guaranteed to be served by the L1 cache, and (2) computing the effective cache-access latency for L1 and LLC on superscalar CPUs. Here, we elaborate on our techniques.

**Cache-hit analysis** The obvious upper bound one can derive for `merge-sort` and `linear-merge` is to count all their memory accesses $M$, assume that all of them will be served by the LLC, and multiply the number of accesses by the latency of LLC accesses from the specification, i.e., $M \cdot l_{L3}$. However, this bound is very conservative because locality of reference suggests only a small fraction of accesses is served by the LLC.

To find a more accurate upper bound, we examined the structure of the two primitives and found a number of accesses that are certain to be served by the L1 cache. In our analysis, we assumed that the primitives are to be run on a machine with an 8-way associative cache and that local variables are memory-aligned in a way that each variable falls in a different cache set, so that they never evict each other. Our analysis yielded the following insights:

1. Both input relations are scanned linearly;
2. All accesses to local variables are served from L1;
3. In every merging iteration, one input data access is the same as the previous iteration and, hence, served from L1;
4. Memory is fetched at cache line granularity, so each L1 miss is followed by $s_{cl}/s_{tuple}$ L1 hits, where $s_{cl} = 64$ is the cache line size in bytes, and $s_{tuple}$ is the size of sorted tuples.

(1) is clearly true because of the way the two primitives work. (2) is true because of the memory alignment of local variables and the fact that there are fewer local variables than there are cache sets, so all of them can fit in L1 and no cache set uses more than one position for local variables. Moreover, since the main loops of `merge-sort` and `linear-merge` touch all local variables, at most two data locations are accessed before the local variables are touched again, and this means no data can evict a local variable in an 8-way associative cache. (3) holds because merging involves two running pointers, and only one of the pointers advances at each iteration. (4) data is aligned in memory, so each L1 miss fetches one cache line to L1, i.e., exactly $s_{cl}/s_{tuple}$ data items. Moreover, both input relations are scanned linearly, and, therefore, for each miss, the following $s_{cl}/s_{tuple} - 1$ accesses will be L1 hits.

Our analysis helped us identify $M_{L1}$, i.e., a number of accesses that are certain to be fetched from L1. Having that, we updated our timing bound model to be $M_{L1} \cdot l_{L1} + (M - M_{L1}) \cdot l_{L3}$. In practice, we found out that a large portion of accesses are certain to be fetched from L1 (about 89.06% and 79.73% of $M$ for `merge-sort` and `linear-merge`, respectively). The bound given by our cache-hit analysis is a safe bound, in the sense that the actual execution time cannot ever exceed the estimated bound.
Calculating effective latencies

Even though the upper bound derived by the cache-hit analysis is much better than our initial approach, it turned out that the estimated execution time was still about an order of magnitude slower than the actual execution. This can be attributed to the fact that modern processors have a highly efficient and parallel pipeline.

To better estimate the execution time of the two primitives, we decided to measure the effective latency $l_{L1}^*$ and $l_{L3}^*$ of L1 hits and misses respectively, and plug them into the formula derived by our cache-hit analysis. To measure these effective latencies we performed several runs of the primitives for different randomly generated input data and collected a series of measurement tuples $(m_{L1}^i, m_{L3}^i, c^i)$, where $m_{L1}^i$ was the number of L1 hits and $m_{L3}^i$ was the number of L1 misses, as reported by the CPU’s performance counters, and $c^i$ was the observed execution time in cycles. Given these tuples, we assumed a linear model $m_{L1}^i \cdot l_{L1}^* + m_{L3}^i \cdot l_{L3}^* = c^i$, and used regression to derive values for $l_{L1}^*$ and $l_{L3}^*$.

The above model assumes that $l_{L1}^*$ and $l_{L3}^*$ are constant across all measurements, but this is not necessarily true. In an effort to account for outlier data that cause significantly different effective latencies, we repeated the same experiment for input data that was already sorted, and data that was in reverse order. The results (Table 7.1) showed that, even though there is some variation in the measured latencies, it is not more than 0.02 cycles for $l_{L1}^*$, and 0.9 cycles for $l_{L3}^*$.

To make sure that our regression-based timing estimate will indeed be an upper bound of real executions, we used $\hat{l}_{L1} = 0.74$ and $\hat{l}_{L3} = 5$ as effective latencies, which are upper bounds on $l_{L1}^*$ and $l_{L3}^*$, respectively. The resulting estimates were much closer to the real execution time (at about 1.96x) than the estimates we got through the cache-hit analysis alone. Experimental results showed that the estimated bounds were never exceeded by the actual execution time, so we believe that they can be considered a safe upper bound. However, for extra security, users of Hermetic could specify the timing bound to be the one from the cache-hit analysis, at some performance cost.
APPENDIX B
OBLIVIOUS PRIMITIVES WITH FAKE ROWS

Section 4 briefly mentions that we had to modify the oblivious primitives from [4] to achieve two goals: (1) allow the primitives to compute the correct result on relations that have fake rows, and (2) provide an oblivious ways of adding a controlled number of fake rows to the output of certain primitives. This Section lists the modifications we had to perform.

B.1 Supporting fake rows

Fake rows in Hermetic are denoted by their value in the isDummy field. Below we list all the primitives we had to modify to account for this extra field. Keep in mind that, whenever our description involves logic with some kind of branch, we take care to replace branches with the CMOV instruction.

groupid: This primitive groups the rows of a relation based on a set of attributes, and adds an incremental id column, whose ids get restarted for each new group. In order for this to work correctly in the face of dummy records, we need to make sure that dummy records do not get grouped with real records. To avoid this, we expand the set of grouping attributes by adding the isDummy attribute. The result is that real records get correct incremental and consecutive ids.

grsum: Grouping running sum is a generalization of groupid, and as such, we were able to make it work with dummy records by applying the same technique as above.

union: Union expands the attributes of each relation with the attributes of the other relation, minus the common attributes, fills them up with nil values, and then appends the rows of the second relation to the first. To make union work with fake rows, we make sure the isDummy attribute is considered common across all relations. This means that the output of unions has a single isDummy attribute, and its semantics are preserved.

filter: To make filter work with fake rows, we need to make sure that user predicates select only real rows. To achieve this, we rewrite a user-supplied predicate $p$ as “(isDummy = 0) AND $p$”. This is enough to guarantee that no fake rows are selected.

join: What we want for join is that real records from the one relation are joined only with real records from the other relation. To achieve this, we include the isDummy attribute to the set of join attributes of the join operation.

groupby: For the groupby primitive, we apply the same technique as for the groupid and grsum - we expand the grouping attributes with isDummy.

cartesian-product: Cartesian product pairs every record of one relation with every record of the other, and this happens even for fake rows. However, we need to make sure that only one instance of isDummy will be in the output relation, and that it will retain its semantics. To do this, we keep the isDummy attribute of only one of the relations, and we update its value to be 1 if both paired rows are real and 0 otherwise.

multiplicity and histogram: These two primitives need to return the corresponding statistics of the real records. Therefore, we make sure to (obliviously) exclude dummy records for the computation of multiplicities and histograms.
B.2 Adding fake rows to the primitive outputs

To enable the introduction of fake records, we alter the primitives \texttt{filter}, \texttt{groupby}, and \texttt{join}. The \texttt{filter} primitive normally involves extending the relation with a column holding the outcome of the selection predicate, obliviously sorting the relation based on that column, and finally discarding any records which do not satisfy the predicate. To obliviously add \( N \) records, we keep \( N \) of the previously discarded records, making sure to mark them as fake. \texttt{groupby} involve several stages, but their last step is selection; therefore, one can add fake rows in the same way. \texttt{join} queries involve computing the join-degree\(^1\) of each record in the two relations. To add noise, we modify the value of join-degree: instead of the correct value, we set the join-degree of all fake records to zero, except one, whose degree is set to \( N \). By the way that joins work, this is enough to eliminate all previous fake records, and create exactly \( N \) new fake records.

---

\(^1\) In a join between relations \( R \) and \( S \), the join-degree of a record in \( R \) corresponds to the number of records in \( S \) whose join attribute value is the same with this row in \( R \).