Hermetic: Privacy-preserving distributed analytics without (most) side channels

Min Xu[‡] Antonis Papadimitriou[†◇] Andreas Haeberlen[†] Ariel Feldman[‡★]

[†] University of Pennsylvania [‡] University of Chicago [★]Google [◇]Duality Technologies

Abstract

Trusted execution environment (TEE), such as Intel SGX, is an emerging technology that enables privacy-preserving distributed analytics on an untrusted cloud platform. TEEs, however, suffer from side channels, such as timing, memory and instruction access patterns, and message sizes that weaken their privacy guarantees. Existing attempts to mitigate these channels in analytics systems are unsatisfactory because they do not attempt to address multiple critical side channels simultaneously and employ mitigations that are inefficient, unprincipled, or difficult to use.

In this paper, we present Hermetic, a data query processing system that offers a principled approach to mitigating four of the most critical digital side channels simultaneously. We introduce an *oblivious execution environment* that supports fast, non-oblivious computations without leakage, as well as fast secure query operators using this primitive to process large data sets. We apply the *differentially private* (DP) padding mechanism on execution time and output sizes of query operators, which avoids prohibitive padding overheads as in most prior solutions, with provable privacy. To achieve efficient DP padding in complex query, we further introduce an *privacy-aware* query planner that can optimize a query plan under user's privacy constraints. Our experimental evaluation of a Hermetic prototype shows that it is competitive with previous privacy-preserving systems, even though it provides stronger privacy guarantees.

1 Introduction

Recently, several systems have been proposed that can provide *privacy-preserving distributed analytics* [79, 99]. At a high level, these systems offer functionality comparable to a system like Spark [96]: mutually untrusted data owners can upload large data sets, which are distributed across a potentially large number of nodes, and they can then submit queries across the uploaded 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 data owner may wish to protect it against a potentially curious or compromised cloud platform.

It is possible to implement privacy-preserving analytics using cryptographic techniques [71, 76], but the resulting systems tend to have a high overhead and can only perform a very limited set of operations. An alternative approach [69, 79, 99] 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 compromised the operating system on the machines that hold the data.

However, even though trusted 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. Currently, the most notorious attacks exploiting such channels are the various Meltdown [52], Spectre [47], and Foreshadow [87] variants, which are due to vulnerabilities in current CPU designs. These channels are devastating because they affect a huge number of deployed CPUs, and, in the case of Foreshadow, even compromise the security of the trusted SGX enclave itself! However, from a scientific perspective, they are actually not the most dangerous ones: now that the underlying vulnerabilities are known, they can be fixed in the next generation of CPUs [41]. In contrast, there are many other side channels – including the execution time of a target program [50], the sequence of memory accesses from the enclave [94], the number and size of the messages that are exchanged between the nodes [68], the contents of the cache [14], and the fact that a thread exits the enclave at a certain location in the code [51, 91] – that are more fundamental and will stay with us even after Meltdown, Spectre, and Foreshadow have been fixed.

Side channel leakage in privacy-preserving analytics has received considerable attention recently [5, 6, 68, 69, 99], but existing proposals still suffer from two main limitations. First, they do not attempt to mitigate the most critical side channels simultaneously or to evaluate the performance impact of doing so. For instance, Opaque [99] explicitly declares timing channels to be out of scope. Second, the mitigations that they do employ are unsatisfying. The most common approach is to pad computation time and message size all the way to their worst-case values. But as we show experimentally, full padding can drive up overhead by several orders of magnitude. Furthermore, existing attempts to avoid full padding suffer from one of the three problems: i) employing ad hoc schemes that lack provable privacy guarantees [68]; ii) relying on users to specify padding bounds a priori, which we believe is unrealistic [99]; and iii) failing to support complex data analysis due to limited expressiveness [1, 46].

This paper proposes Hermetic, which aims to address all of these limitations. Hermetic mitigates the four major digital side channels – timing, memory access patterns, instruction traces, and output sizes – a challenging design problem in itself, while achieving performance that is as good or better than systems with weaker privacy guarantees. Moreover, Hermetic combines a method of padding the execution time and result size of database queries that is substantially more efficient than full padding and a new privacy-aware query planner to provide principled privacy guarantees for complex queries, i.e., those with multiple join operations, and not unreasonably burdening the user.

To achieve these goals, we employ a three-pronged strategy either to optimize the critical components or to fill in the missing parts in previous solutions. The first part is a primitive that can perform fast, non-oblivious sort securely, by "locking-down" core and carefully transforming the program to achieve several key properties. This primitive, namely *oblivious execution environment (OEE)*, protects against digital side channels and yet improves efficiency. Importantly, we also provide a concrete implementation of an OEE: although related primitives have been discussed previously (e.g., in [24, 99]), existing work either simply assumes it without details or achieves weaker security definitions than OEE does (e.g., in [18, 58].

The second element of our approach is a set of oblivious query operators that are faster and more secure than traditional oblivious operators. In particular, we enable the first secure inner equi-join that is resilient to the four major side channels, and avoids the prohibitive overheads from full padding by relaxing the privacy guarantee on output size channel to differential privacy [22]. The third element is a privacy-aware query planner that generates an efficient query plan that respects the user's specification on privacy and performance. Differential privacy introduces a set of privacy parameters that affect both the overall performance and privacy cost of the query plan to the query optimization problem, and we solve the problem over the entire privacy parameter space, as defined by user's input, while existing solutions have to assume a fixed set of privacy parameters, which misses huge optimization opportunity.

We have implemented a Hermetic prototype that runs on commodity machines. Since 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 today's equipment. Our experimental evaluation shows that our approach is indeed several orders of magnitude more efficient than full padding, which has been the only principled side channel mitigation in analytics systems against the timing and output side channels. At the same time, we show that Hermetic has comparable performance to existing SGX-based analytics systems while offering stronger privacy guarantees.

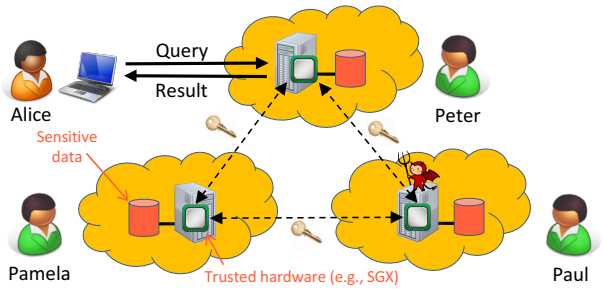


Figure 1. Example scenario. Analyst Alice queries sensitive data that is distributed across multiple machines, which are potentially owned by multiple participants. An adversary has complete control over some of the nodes, except the CPU.

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 implemented correctly and has not been compromised. Also, there are physical side channels that even Hermetic cannot plug: for instance, an adversary could use power analysis [48] or electromagnetic emanations [49], or simply de-package the CPU and attack it with physical probes [81]. These attacks are more difficult and expensive to carry out than monitoring digital side channels with software.[1] Thus, despite these caveats, Hermetic brings privacy-preserving distributed analytics closer to practicality. Our contributions are as follows:

- the design of the Hermetic system (Section 3),
- the OEE primitive, which performs fast, non-oblivious computations privately on untrusted host (Section 4),
- enhanced oblivious operators, including a new oblivious join algorithm with DP padding, that limit four different side channels (Section 5),
- a novel privacy-aware query planner (Section 6),
- a prototype of Hermetic (Section 7), and
- a detailed experimental evaluation (Section 8)

2 Overview

Figure 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, which we call *federated data analysis*. 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 and the input datasets are – and that each node contains a

[1] They may also be impossible to eliminate without extensive hardware changes. Yet, as physical side channel attacks often aim to infer control flow, we speculate that Hermetic's mitigations of the instruction trace side channel may help mitigate these physical channels as well.

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 [79, 99], which assumes that there is only one participant, who outsources a data set to a set of nodes, e.g., in the cloud.

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, except the CPU; she can run arbitrary software, make arbitrary modifications to the OS, and read or modify any data that is stored on these nodes; she can probe the memory bus between CPU and the memory to get the memory traces. This threat model inherently addresses all previous user-space side channel attacks [30, 31, 74, 95]. 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* [22]. 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, δ)-differentially private if, for all possible sets of outputs $S \subseteq R$ and all similar input data sets d_1 and d_2,

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

That is, with probability at least $1 - \delta$, 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, but require more random noise to output. For more information on how to choose ε and δ, see, e.g., [35].

Differential privacy has strong composition theorems; in particular, if two queries q_1 and q_2 are $(\varepsilon_1, \delta_1)$- and $(\varepsilon_2, \delta_2)$-differentially private, respectively, then the combination $q_1 \cdot q_2$ is $(\varepsilon_1 + \varepsilon_2, \delta_1 + \delta_2)$-differentially private [21, 22]; note that it does not matter what specifically the queries are asking. 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 $\sum_i \varepsilon_i \leq \varepsilon_{\max}$. The latter is only a lower bound; the differential privacy literature contains far more sophisticated composition theorems [23, 44, 63, 78], and these can be used to answer far more queries using the same privacy "budget". However, to keep things simple, we will explain Hermetic using only simple composition.

2.2 Straw-man solution with TEEs

It may appear that the problem we motivated above could be solved as follows: 1) each participant leverages secure enclave and remote attestation to protect the confidentiality and integrity of data and computations on other participants, which is similar to the encrypt mode in Opaque [99]; 2) participant, P_i, sets a local privacy budget $\varepsilon^0_{\max,i}$ for dataset d_i, and, for the t-th query, only *query plan* with valid ε_i – that is, $\varepsilon_i \leq \varepsilon^t_{\max,i}$, on each dataset d_i is allowed to execute, and then $\forall i,\ \varepsilon^{t+1}_{\max,i} = \varepsilon^t_{\max,i} - \varepsilon_i$.

This approach would seem to meet our requirements: differential privacy ensures that a malicious analyst cannot compromise privacy, and the enclaves ensure that malicious participants cannot get access to intermediate results and/or sensitive data from other nodes, even when the system must send such data to their own nodes as part of the query (e.g., to be joined with some local data).

2.3 Problem: Side channels

However, the straw-man 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 devastating ones, first identified by Tople and Saxena [85], are:

- **Timing channel (TC) [50]**: The adversary can measure how long the computation in the enclave takes to infer the execution path, which leaks sensitive information if branches or indirect jumps depend on the data; and
- **Memory channel (MC):** The adversary can observe the locations in memory that the enclave reads or writes (even though the data itself is encrypted!), which leaks sensitive information if there are memory accesses indexed by the data; and
- **Instruction channel (IC):** The adversary can see the sequence of instructions that are being executed, e.g., by probing the branch target buffer [51], to infer the execution path, as in TC; and
- **Object size channel (OC):** The adversary can see the size of any intermediate results that the enclave stores or exchanges with other enclaves, and these size measurements could be used to reconstruct substantial information about the data [68].

In general, the connectivity pattern between the enclaves could be another channel, but in our setting, this pattern is a function of the (non-sensitive) query and not of the (sensitive) data, so we do not consider it here.

At first glance, the above 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 [98].

To get truly robust privacy guarantees, it is necessary to close or at least mitigate these channels.

2.4 State of the art

Most prior techniques to address side channels in data analysis have either used heuristics to reduce the bandwidth of some of the channels, as, e.g., in [7, 36, 45], or they have strived to completely remove dependencies between the data and the channels that an attacker can observe. The former approach is unsatisfying, since it is difficult to formally reason about the true strength of its guarantee, whereas the latter comes at great cost. For instance, *data oblivious algorithms* [6] propose algorithms that address the MC by accessing the data in a manner that depends only on data's size and not on its content. These application-specific algorithms can achieve better performance than generic ORAM (and they are one of Hermetic's building blocks), but they are still quite inefficient. Furthermore, *full padding* can eliminate the TC by padding computation time to its worst-case value, but it potentially comes with orders of magnitude of overhead.

More recent work [1, 46] has adopted cheaper mitigations, including using differential privacy to determine the amount of padding needed to obscure data sizes and access patterns. But, it has yet to address more complex relational operations, such as joins, or how to integrate DP into query planning.

2.5 Approach

Our goal in this paper is to get the "best of both worlds": we want to mitigate the four side channels from Section 2.3 while achieving performance that is as good or better than prior privacy-preserving analytics systems.

Our strategy for doing so benefits from a key observation: *OC leakage is similar to leakage through query results.* For example, observing the size of a selection query's result set is the same as to being given the results of a `select count` query. Thus, as we will show, it is possible to use differential privacy to determine the amount of padding to add to query operators in order to mitigate side channels even for more complex operations like joins. Whereas DP applied to query results provides principled techniques for adding noise to results while preserving utility, DP applied to side channels offers a principled means for adding dummy tuples to relations while preserving efficiency. Our system consists of three components:

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 $|in|$ and f, but not on the actual data values. This mitigates all four channels.

Enhanced oblivious operators: We present several enhanced oblivious operators, including a new oblivious join algorithm with DP padding, that can be used to compose OEE invocations into complex query plans. In addition to avoiding MC – like all oblivious algorithms – our operators also have a deterministic control flow (IC & TC), they use only timing-stable instructions (TC), and the size of their output is either constant or noised with "dummy tuples" to ensure differential privacy (TC & OC).

Privacy-aware query planner: We describe a query planner that optimizes for both efficiency *and* privacy. It allows users to assign weights to and bounds on the performance of query execution and the privacy cost to each relation. It then uses *multi-objective optimization* [86] to generate an efficient query plan, annotated with privacy budget ε on each operator, that respects these priorities.

3 The Hermetic System

In this section, we describe the workflow of federated query processing in Hermetic, and defer the detailed discussions on each of the key components, as well as the prototype implementation, to Sections 4, 5, 6, and 7

3.1 Overview

Hermetic consists of a master node, and several worker nodes. These nodes can switch roles for data proximity, load balance or policy regulations. Each node runs the trusted hypervisor to support OEEs (Section 4), and the trusted runtime, inside a TEE, that includes the Hermetic operators (Section 5) and a light-weight plan verifier. The last component of Hermetic is the untrusted query planner (Section 6). The workflow of Hermetic consists of the following steps:

1. Initially, the master node launches the hypervisor and runtime, and the data owners contact the runtime to setup master encryption keys and upload their data (Section 3.2). Data owners verify the authenticity of both the hypervisor and the runtime via attestation (Section 3.3).
2. After initialization, analysts can submit queries to the query planner, which generates a concrete query plan and forwards it to the runtime (Section 3.4).
3. As the planner is outside trusted computing base, the runtime verifies incoming plans to make sure that all operators are annotated with the appropriate sensitivities and ε's (Section 3.4).
4. If verification passes, the runtime organizes the worker nodes to execute the query plan using Hermetic's oblivious operators. (Section 3.5).

We describe these steps in greater detail below.

3.2 Initialization

Hermetic is initialized after the data owners set up master encryption keys and upload their sensitive data to the server. Since no party in Hermetic is completely trusted, the master keys are created inside the trusted runtime, using randomness contributed by the data owners. After that, the keys are

encrypted using a hardware-based key and persisted to secondary storage using, e.g., SGX's sealing infrastructure [3].

With the master key in place, data owners send their data, together with the associated privacy budgets, to the runtime, which encrypts it with the key and stores it to the disk.

3.3 Attestation

A prerequisite for uploading sensitive data is that data owners 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 Hermetic runtime is running in a TEE. We achieve this level of trust in two stages. First, upon launch, the Hermetic runtime uses a mechanism such as Intel's trusted execution technology (TXT) [39] to get an attestation of the code loaded during the boot process. If the hypervisor is absent from the boot process, the Hermetic runtime halts. Second, data owners leverage enclave attestation, e.g., Intel SGX attestation [3], to verify that the correct runtime is running in the TEE.

3.4 Query submission and verification

Analysts write their queries in a subset of SQL that supports `select`, `project`, `join`, and `groupby` aggregations. Analysts can supply arbitrary predicates, but they cannot run arbitrary user-defined functions. Analysts submit queries to the query planner, which is outside Hermetic's TCB. The planner then prepares a query plan to be executed by the runtime.

As explained in Section 6, query plans are annotated with the sensitivity of each relational operator, as well as with the ε for adding noise to the intermediate results. Since the planner is not trusted, these parameters have to be verified, so that the enough amount of noise will be added for the required privacy, before the plan is executed: Hermetic has to check that the sensitivities are correct by computing them from scratch based on the query plan, and that the total ε annotations do not exceed the privacy budgets. δ is a system parameter enforced by Hermetic runtime, and it is not explicitly annotated or verified in the plan (Section 5.2). Correctness and efficiency are out of the scope of Hermetic's verification.

The untrusted platform could launch a rollback attack [11, 13, 56, 84], in which it tricks the trusted runtime into leaking too much information by providing it with a stale copy of the privacy budget. To ensure the freshness of the stored privacy budget, the runtime must have access to a protected, monotonically-increasing counter. This counter could be implemented using a hardware counter, such as the one optionally available with SGX [38] – possibly enhanced with techniques to slow wear-out of the counter [11, 84]. Alternatively, it could be implemented with a distributed system consisting of mutually-distrusting parties [13, 56].

3.5 Query execution

If a plan passes the verification, it is executed by the runtime. Before execution starts, the privacy budget on each relation is decreased based on the εs in the plan, and the runtime generates the Laplace noise which determines the number of dummy tuples 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 5 to perform the computation.

4 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).*

4.1 OEE properties

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 has access to TC, MC, IC and OC.

To provide a solid foundation for oblivious execution without performing dummy memory accesses, we introduce a primitive OEE (f,in,out) that, for a small set of predefined functions f, has the following four properties:

1. Once invoked, OEE runs to completion and cannot be interrupted or interfered with;
2. OEE loads *in* and *out* into the cache when it starts, and writes *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, $|in|$, and $|out|$; 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 *in* and *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 sequentially, 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 OEE is allowed to perform data-dependent memory accesses *during* its execution. Effectively, OEE is allowed to use a portion of the CPU cache as a private, unobservable memory for the exclusive use of f. This is what enables Hermetic to provide good performance.

4.2 Challenges in building an OEE today

Prior work has recognized the performance benefits of having an un-observable environment for performing data-dependent functions. Zheng et al. [99] speculate that a future un-observable memory would allow a system to perform ordinary quick-sort instead of expensive oblivious sort. Unfortunately, actually realizing an OEE, especially on current hardware, is challenging.

We can achieve property #3 by eliminating data-dependent branches and by padding the execution time to an upper bound via busy waiting (Section 7.2). We can also disable hardware features such as hyper-threading 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 (Sections 4.3, 7.2). Finally, by executing the OEE in a TEE, e.g., SGX, enclave, we can ensure that the data is always encrypted while in memory.

However, even if we ignore the vulnerabilities from [87], today's TEE, e.g., SGX, *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 [17]. Indeed, these limitations have already been exploited to learn secret data inside enclaves [14, 51, 94]. Flicker [58] achieves property #1 by suspending the entire machine, except the sensitive program, using the SKINIT instruction, and we want to achieve property #1 without suspending concurrent processes.

4.3 The Hermetic OEE

Realizing an OEE requires that f in OEE be adapted with deterministic instruction sequences and constrained memory footprints, e.g., by avoiding recursion. Algorithm 1 shows how we achieve the oblivious instruction trace, in terms of op-codes, for the core `merge-sort` function. We unify all the conditional branches, including those depending on the data values and the operation mode, into one execution path using the `cwrite` primitive. Hermetic actually optimizes Algorithm 1 so that the sorted order is kept in the global `oee_buffer` after `merge-sort`-ing on the sorting attributes, and the rest of the attributes are linearly re-ordered following the sorting order, without the cost of `merge-sort`.

Furthermore, the CPU core that the OEE is running on must be configured so that no other processes can interrupt it or interfere with its state. To achieve the latter, Hermetic relies on a thin layer of hypervisor to configure the underlying hardware for the isolation, as shown in Algorithm 2. 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) disables speculation across the OEE boundary to mitigate Spectre-style attacks by setting the appropriate CPU flags or using serialization instructions [41]; (3) configures Intel's Cache Allocation Technology (CAT) [65] to partition the cache between the OEE's core and the other cores at the hardware level; (4) prevents the OS from observing the OEE's internal state by accessing hardware features such as performance monitoring; (5) flushes legacy data from previous executions in the isolated cache partition; and (6) when the function completes, restore the hardware configuration, and flushes the legacy state of the OEE. In Section 7.1, we present further details of the hypervisor's design and its use of the CAT.

Second, the program in OEE should be prefixed with sanitization to preload all program instructions and memory states into the isolated cache partition, and postfixed with cleansing to deterministically pad the execution time and flush the cache-lines. In Section 7.2, we present further details of the preloading and time padding. Note that one OEE invocation can process up-to `OEE_SIZE` tuples, and the total size is bounded by the last-level cache partition. We describe how to build large-scale operators in Section 5.

We view the hypervisor as interim step that makes deploying Hermetic possible today. In terms of security, the Hermetic hypervisor is equivalent to the *security monitor* in [18], and they both have small TCBs that can be formally verified and attested to. Its functionality is constrained enough that it could be subsumed into future versions of TEE, e.g., 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.

5 Oblivious operators

OEEs provide a way to safely execute simple computations, e.g., sorting, on blocks of data, while mitigating side channels. However, to answer complex queries over larger data, Hermetic needs higher-level operators, as described next.

Our starting point is the set of so-called oblivious query operators, introduced in prior work [6, 69], whose memory access patterns depend only on their input size, not the specific values. These include inherently oblivious relational operators, such as `project`, `rename`, `union` and `cartesian-product` as well as operators based on oblivious sorting networks (e.g., Batcher's odd-even merge-sort [9] — `batcher-sort`). Oblivious sort is the basis for auxiliary oblivious operators like `group-running-sum`, `filter`, `semijoin-aggregation` and `expand`, which in turn can be combined to form familiar relational such as `select`, `groupby`, `orderby` and `join`. In particular, an oblivious inner equi-join could be constructed by first applying `semijoin-aggregation` [6] on the two input relations to derive the *join degree*, namely the number of matches for a tuple from the other relation, then `expand`-ing the two relations based on join degree following *equally-interleaved expansion* [69], obliviously sorting the expanded relations ordered by join attributes and expansion id, and, finally, `stitch`-ing them together to get the result. Note that , before Hermetic, there is no existing system that supports

Algorithm 1: OEE `merge-sort` with data oblivious instruction trace. `end`, `ascend`, `attrs` are function parameters pre-loaded into global `oee_buffer`, and `tb` points into the `oee_buffer` for storing tuple input and output. `OEE_SIZE` indicates the maximum number of tuples that one OEE invocation could process.

```
1  func merge − sort(mode)
2    for len ∈ {2^0, ..., 2^log(end)} do
3      for off ∈ {0, 2·len, ..., ⌊end/(2·len)⌋ · 2 · len} do
4        right ← MIN(off + 2 · len − 1, end)
5        pos_1 ← off; pos_2 ← off + len
6        for cur ∈ {0, ..., right − off} do
7          cond_a ← (pos_1 ≤ off + len − 1)
8          cond_b ← (pos_2 ≤ right); cond_c ← 0
9          cwrite(cond_a, cur_1, pos_1, off + len − 1)
10         cwrite(cond_b, cur_2, pos_2, right)
11         foreach f ∈ attrs do
12           cwrite((cond_c = 0), cond_c, tb[cur_1][f] −
               tb[cur_2][f], cond_c)
13         cond_d ← ((cond_c ≤ 0) = ascend)
14         cond_1 ← (mode = PRELOAD)?(pos_1 <=
             (pos_2 − len)) : (cond_a · cond_d > cond_b − 1)
15         for f ∈ {0, ..., FIELDS_PER_RUN} do
16           cwrite(cond_1, tb[OEE_SIZE + off +
               cur][f], tb[cur_1][f], tb[cur_2][f])
17         cwrite(cond_1, pos_1, pos_1 + 1, pos_1)
18         cwrite(cond_1, pos_2, pos_2, pos_2 + 1)
19       cwrite((mode = PRELOAD), l, 0, OEE_SIZE)
20       for cur ∈ {0, ..., right − off} do
21         tb[off + cur] ← tb[off + l + cur]
22 func cswrite(cond, out, in_1, in_2)
23   asm volatile("test eax, eax"
24     "cmovnz %2, %0", "cmovz %3, %0"
25     : "=r"(*out) : "a"(cond), "r"(in_1),
        "r"(in_2) : "cc", "memory")
```

data-oblivious inner equi-join for federated data analysis. See Appendix A.1 for more details about these operators.

5.1 Extending oblivious operators

Existing oblivious query operators are vulnerable to TC and IC. We eliminate the data-dependent branches by unifying them into one execution path using the `cwrite` primitive (Algorithm 1). We avoid instructions with data-dependent timing following [4].

Furthermore, we accelerate existing oblivious operators, by replacing `batcher-sort` with `hybrid-sort` that leverages OEEs (See Algorithm 3). `hybrid-sort` is faster than `batcher-sort` because each block data in OEE is sorted by faster `merge-sort` (Line 3). This would accelerate the `select`, `groupby` and `join` whose performance is dominated by oblivious sort.

Algorithm 2: OEE isolation.

```
1  func oee − sort(R, attr, order)
2    Disable preemption/interrupts
3    Set speculative execution boundary
4    Configure CAT for last-level cache isolation
5    Disable PMC read
6    Flush the entire cache partition
7    Load attr, order into global oee-buffer
8    foreach FIELDS_PER_RUN attrs ∈ R do
9      Load the attributes of R to oee-buffer
10     Linear scan over oee-buffer   // cache data
11     merge-sort(PRELOAD)           // cache code
12     merge-sort(REAL)
13     Copy sorted attributes from oee-buffer to R
14   Pad the execution time
15   Flush cache & restore H/W configurations
```

Algorithm 3: The OEE-assisted `hybrid-sort` primitive.

```
1  func hybrid − sort(R = {t_0, ..., t_n}, attr, order)
2    if |R| ≤ OEE_SIZE then
3      oee − sort(R, attr, order)
4    else
5      hybrid − sort({t_0, ..., t_{n/2}}, attr, order)
6      hybrid − sort({t_{n/2+1}, ..., t_n}, attr, order)
7      hybrid − merge(R, attr, order)
8  func hybrid − merge(R = {t_0, ..., t_n}, attr, order)
9    if |R| ≤ OEE_SIZE then
10     oee − merge(R, attr, order)
11   else
12     hybrid − merge({t_0, ..., t_{n/2}}, attr, order)
13     hybrid − merge({t_{n/2+1}, ..., t_n}, attr, order)
14     for i ∈ {2, 4, ..., n − 2} do
15       toSwap ← ((t_i[attr] ≤ t_{i+1}[attr]) = order)
16       cwrite(toSwap, t'_i, t_{i+1}, t_i)
17       cwrite(toSwap, t'_{i+1}, t_i, t_{i+1})
18       t_i ← t'_i; t_{i+1} ← t'_{i+1}
```

Finally, enabling Hermetic's query planner to trade off privacy and performance (Section 6) requires obliviously collecting statistics about the input data. To do so, we introduce two new primitives that leverage OEEs: `histogram` that computes a histogram over the values in a given attribute, and `multiplicity` that computes the multiplicity of a attribute – i.e., the number of times that the most common value appears.

5.2 Differentially-private padding

Hermetic adopts an efficient approach to mitigating OC effectively, without padding the number of output tuples of an intermediate query operator to its worst-case value. It

determines the amount of padding to add based on a truncated, shifted Laplace mechanism that ensures non-negative noise size. In particular, for an operator O_i with estimated sensitivity s_i – the maximum change in the output size that can result from adding or removing one input tuple, and the privacy parameter ε_i, $\Delta \sim Lap(o_i, s_i/\varepsilon_i)$, where o_i indicates the offset of the shifted Laplace distribution from 0, dummy tuples are added to the output if $\Delta \geq 0$; Otherwise, 0 dummy tuple is added. This mechanism provides (ε, δ)-differential privacy [21], where δ corresponds to the probability of truncation – i.e., $\delta_i = P_{trunc} = Pr[Lap(o_i, s_i/\varepsilon_i) < 0]$. In addition, P_{trunc} is configured, as a system parameter in Hermetic, with very small value to minimize leakage. Note that the idea of applying differentially private padding to OC is not new, and has been investigated in [1, 46]. But Hermetic enables DP padding on operators, such as inner equi-join, that are considered as future work of [1, 46]. Furthermore, the problem of optimizing a plan, the outputs of whose operators are padded using DP, for both privacy and efficiency is a challenging problem, and Hermetic introduces a new privacy-aware query planner to address this challenge (Section 6).

To implement this approach, relations must be padded with *dummy tuples* to hide the true size of query results. As in prior work, we identify dummy tuples by adding an additional `isDummy` attribute to each relation, adapt operators like `select`, `groupby`, and `join` to add dummy tuples to their results, and adapt query predicates to ignore tuples where `isDummy == TRUE` (See Appendix C for details). Moreover, the actual noise value must be kept hidden from an adversary. As a result, the number of dummy tuples has to be sampled in an enclave, and the sampling process must be protected from side channels, especially timing [4].

6 Privacy-aware Query Planning

In this section, we describe how Hermetic assembles the operators from the previous section 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 a twist: Hermetic is free to choose the amount of privacy budget ε it spends on each operator. Thus, it is able to make a tradeoff between privacy and performance: smaller values of ε result in stronger privacy guarantees but also add more dummy tuples, which slows down subsequent operators.

Query planning in Hermetic follows the same design as in [**?**], and Figure 2 illustrates the key query planning steps for a counting query over three datasets:

Sensitivity estimation: For each possible execution plan tree , Hermetic query planner derives the upper bound on the sensitivities of all the operators O_i in the plan. To do this, the untrusted query planner could initiate auxiliary queries, which we call *leakage queries*, to compute the number of tuples in each operator's output. For instance, in Figure 2, the

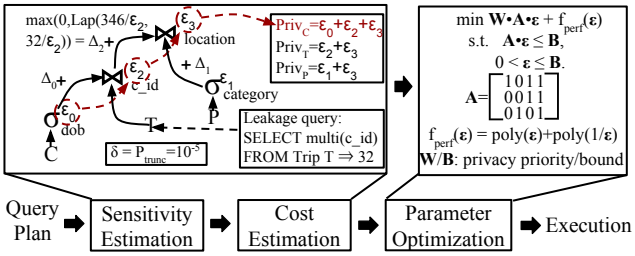


Figure 2. Query planning for query "`SELECT count(*) FROM C, T, P WHERE C.cid=T.cid AND T.location = P.location AND C.age≤27 AND P.category='hospital'`".

planner uses a leakage `select` query with the `multiplicity` operator on the joined attribute, `c_id`, to get an upper bound on the sensitivity of the leftmost join. The leakage queries are differentially private just like ordinary queries, and their (small) cost is charged to the (ε, δ) budget as usual; thus, the planner does not need to be trusted.

Cost estimation: Hermetic planner estimates the symbolic privacy and performance costs of the plan. The privacy cost of a plan, each operator O_i of which is assigned with privacy budget ε_i, is simply $(\sum_i \varepsilon_i)$, but estimating the performance cost is more challenging. 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 1. Hermetic uses the `histogram` operator to estimate the size of intermediate results, following [34, 75]. These estimates are used only to predict the performance of the query plans; thus, if the adversary were to carefully craft the data to make the estimate inaccurate, the worst damage they could do would be to make Hermetic choose a slower query plan. To enable the planner to assess the performance implications of the dummy tuples, Hermetic takes them into account when estimating relation sizes, and the expected number of dummy tuples added by operator O_i, following the distribution $Lap(o_i, s_i/\varepsilon_i)$, is simply the offset o_i.

Parameter optimization: Hermetic's query planner uses multi-objective optimization [86] to find the optimal query plan that matches the user's priorities. In particular, the user specifies a vector of bounds, $\mathbb{B}$, and a vector of weights, $\mathbb{W}$, for the privacy costs on all the input relations. The planner outputs the plan, whose weighted sum of all the costs, including privacy and performance whose weight is always 1, is optimal, and all of the privacy costs are within the bounds. We leave the details on the optimization technique in Appendix D due to space limit.

7 Implementation

To evaluate our approach, we built a prototype, which we now describe, focusing on the hypervisor and OEEs.

Table 1. Cost model for Hermetic's relational operators. n: first input size; c: OEE capacity; m: second input size; k: output size.

Operator	Cost
`hybrid-sort`	$h(n) = n \cdot \log(c) + n \cdot \log^2(n/c)$
`select`	$h(n)$
`groupby`	$3 \cdot h(n)$
`join`	$4 \cdot h(n+m) + 2 \cdot h(m) + 3 \cdot h(n) + 2 \cdot h(k)$

7.1 Hermetic hypervisor

Hermetic's hypervisor extends Trustvisor [57], a compact, formally verified [89] hypervisor. To enable OEEs to "lock down" CPU cores, we added two hyper-calls — `LockCore` and `UnlockCore` — to Trustvisor. `LockCore` works as follows: (1) it checks that hyper-threading and prefetching are disabled (by checking the number of logical and physical cores using `CPUID`, and using techniques from [90]), (2) it disables interrupts and preemption (3) it disables the `RDMSR` for non-privileged instructions to prevent snooping on package-level performance counters of the OEE's core, (4) it flushes all cache lines (with `WBINVD`[2]), (5) it uses CAT [66] to assign a part of the LLC exclusively to the OEE core. `UnlockCore` reverts actions 2–5 in reverse order. These changes required modifying 300 SLoC of Trustvisor.

7.2 Oblivious execution environments

Recall that the goal of OEEs to is create an environment where a carefully-chosen function can perform data-dependent memory accesses that are unobservable to the platform. To achieve this, we must make the function's memory access patterns un-observable and its timing predictable.

Un-observable memory accesses: To make memory accesses un-observable, we ensure that all reads and writes are served by the cache: we disable hardware prefetching and preload all data and code into the cache prior to executing the function. To preload data, we use `prefetcht0`[2], which instructs the CPU to keep the data cached, and we perform dummy writes to prevent leakage through the cache coherence protocol. To preload code, the function first executes in PRELOAD mode to exercise all the code — loading it into the icache — but processes the data deterministically. We align all buffers, especially `oee_buffer`, to avoid cache collisions.
Predictable timing: To ensure that an OEE function's execution time is predictable regardless of the input data, we employ a three-pronged approach. First, we statically transform the function's code to eliminate data dependent control flow (See Algorithm 1) and instructions with data-dependent timing. Second, although we allow the function's memory accesses to be data-dependent, we carefully structure it to constrain the set of possible memory accesses, thereby making the number of accesses that miss the L1 cache, and thus must be served by slower caches, predictable. For example, `merge-sort` (See Algorithm 1) performs a single pass over the input and output buffers to ensure that there are few cache misses per iteration. Finally, to account for timing variation that might occur in modern superscalar CPUs (e.g., due to pipeline bubbles), we pad the OEE's execution time to a conservative upper bound that is calibrated to each specific model of CPU. This bound is roughly double the execution time and was never exceeded in our experiments. For more information, see Appendix B.

[2]Note that the timing variations of `clflush` and `prefetch` that are exploited in [30, 31] are not a problem for Hermetic simply because the memory addresses, during flushing and preloading, are observable, in a deterministic manner, to the adversary anyway.

Table 2. Experimental configurations and their resilience to different channels (`MS`: `merge-sort`, `BS`: `batcher-sort`, `HS`: `hybrid-sort`, `CP`: cartesian product, `SMJ`: sort-merge join).

Configuration	Sort	Join	MC	IC	TC	OC
NonObl	`MS`	`SMJ`	✗	✗	✗	✗
DOA-NoOEE	`BS`	[6, 69]	✓	✗	✗	✗
Full-Pad	`BS`	`CP`	✓	✓	✓	✓
HMT I	`HS`	Section 5	✓	✓	✓	✗
HMT II	`HS`	Section 5	✓	✓	✓	✓

Table 3. Schema and statistics of the relations. Synthetic was generated with a variety of tuples and multiplicities.

Query	Relation	Tuples	Multiplicities
S1-3	Synthetic	*	*
Q4-6	Trips	$10M$	m(cid)=32, m(loc.)=1019
	Customers	$4 \cdot 1M$	m(cid)=1
	Poi	$0.01M$	m(loc.)=500
BDB1-3	rankings	$1.08M$	m(url)=1
	uservisits	$1.16M$	m(url)=22

7.3 Trusted computing base

Hermetic's TCB consists of the runtime (3,995 SLoC) and the trusted hypervisor (14,095 SLoC). The former may be small enough for formal verification, and the latter has, in fact, been formally verified [89] prior to our modifications. Recall that the hypervisor would not be necessary with a future TEE that natively supported "locking down" CPU cores.

8 Evaluation

This sections presents the results of our experimental evaluation of Hermetic's security and performance, a comparison with Opaque, the current state-of-the-art, and a discussion of the ability of Hermetic's query planner to trade off privacy and performance.

8.1 Experimental setup

Very recently, Intel has started offering CPUs that support both SGX and CAT; however, we were unable to get access to one in time. We therefore chose to experiment on an Intel Xeon E5-2600 v4 2.1GHz machine, which supports CAT, with 4 cores, 40 MB LLC, and 64GB RAM. This means that the numbers we report do not reflect any overheads due to

encryption in SGX, but, as previous work [99] 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 H/W prefetching because they can cause timing variations.

Table 2 shows the different system configurations we compared, and the side channels they defend against. NonObl corresponds to commodity systems that take no measure against side-channels; DOA-NoOEE uses data-oblivious algorithms from previous work [6, 69], without any un-observable memory; Full-Pad pads output of all operators to the maximum values: pads output of `SELECT` to the input size, and pads output of `join` to the product of the two input sizes using Cartesian join; and HMT I and HMT II implement the techniques described in this paper – the only difference being that the former does not add noise to the intermediate results.

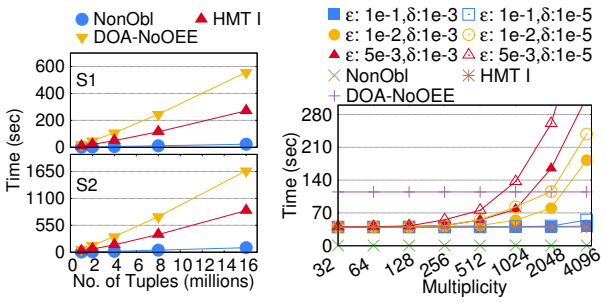
Table 3 lists all the relations we used in our experiments. The `Trips` relation has 5-days-worth of records from a real-world dataset with NYC taxi-trip data [67]. This dataset has been previously used to study side-channel leakage in MapReduce [68]. Since the NYC Taxi and Limousine Commission did not 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 attribute 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 attributes, except those that control the statistics in question. We use the `rankings` and `uservisits` from Big Data Benchmark (BDB) [2] for the comparison with Opaque.

8.2 OEE security properties

To verify the obliviousness of `merge-sort` (`MS`) and `linear-merge` (`LM`), we created synthetic relations, populated with random values and enough tuples to fill the available cache (187, 244 tuples of 24 bytes each).

First, we used the Pin instrumentation tool [55] to record instruction traces and memory accesses; as expected, these depended only on the size of the input data. Second, we used Intel's performance counters to read the number of LLC misses[3] and the number of accesses that were served by the cache[4]; as expected, we did not observe any LLC misses in any of our experiments. Finally, we used `objdump` to inspect the compiler-generated code for instructions with operand-dependent timing; as expected, there were none. More details can be found in Appendix B.1.

[3]Using the `LONGEST_LAT_CACHE.MISS` counter.
[4]Using the `MEM_UOPS_RETIRED.ALL_LOADS` and `MEM_UOPS_RETIRED.ALL_STORES` counters.



(a) S_1&S_2 (b) S_3

Figure 3. Performance of `select` (S_1), `groupby` (S_2) and `join` (S_3) for different data sizes and join multiplicities.

Next, we verify the timing obliviousness of `MS` and `LM` in OEE using cycle-level measurement. as expected, the execution time without padding could vary by tens of thousands of cycles between data sets, but with padding, the variations were only 44 and 40 cycles, respectively. (See Appendix B.3) As Intel's benchmarking manual [70] suggests, this residual variation can be attributed to inherent inaccuracies of the timing measurement code.

8.3 Performance of 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 tuples of a relation that satisfy a predicate, S_2 groups a relation by a given attribute 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 NonObl, DOA-NoOEE , and Hermetic to execute queries $S_1 - S_3$ on these relations.

Figure 3(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 HMT 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 NonObl, but it seems to be an acceptable price to pay for protection against side channels, at least for some applications that handle sensitive data. In comparison to DOA-NoOEE , HMT 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 HMT II for query S_3 on relations of different multiplicities. The amount of noise added to the output in order to achieve differential privacy guarantee is proportional to $-s/\varepsilon \cdot ln(2\delta)$, and sensitivity s is equal to the maximum multiplicity of the join attribute in the two relations. Figure 3(b) shows the performance of HMT II with various ε and δ values, compared to other configurations, and differential privacy only affects the

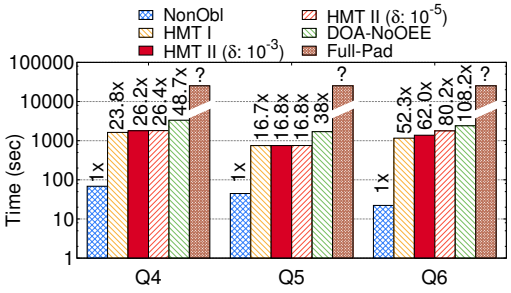


Figure 4. Performance of all configurations for Q_4 – Q_6.

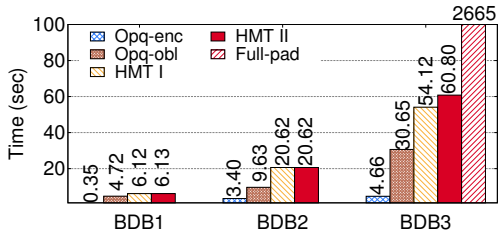


Figure 5. Comparison with Opaque.

overall performance for very small ε, δ and large multiplicity (around 200). In addition, the line for Full-Pad is missing as it cannot finish the query within 7 hours due to its huge padding overheads for tackling OC.

8.4 End-to-end performance

We compared the different system configurations on complex query plans, each of which consists of at least one `select`, `groupby`, and `join` operator. To perform this experiment, we used the relations described in Table 3, 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 4. For HMT II, the system optimized for performance, given a constraint $\varepsilon_{max} \leq 0.05$ on the privacy budget, and we set $\delta = P_{trunc}$ as 10^{-3} and 10^{-5} (Section 5.2). Full-Pad 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, HMT II, which pads using differential privacy, has only a small overhead relative to non-padded execution (HMT 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 (DOA-NoOEE), *even though it offers stronger guarantees.*

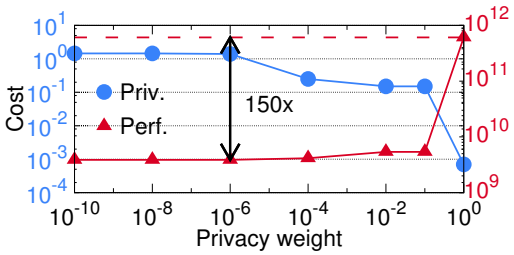


Figure 6. Total privacy and performance costs at various points in the trade-off space.

8.5 Comparison with the state-of-the-art

We compare Hermetic with the state-of-the-art privacy-preserving analytics system, Opaque [99] using the big data benchmark (BDB) under the same configuration as in the previous experiments. In particular, we measure the execution times of the first three queries in BDB (BDB1-3) on the first 3 and 2 parts of the `1node` version of the `rankings` and `uservisits` datasets, respectively. Because the Hermetic prototype only supports integer attributes, we replace all `VARCHAR` and `CHAR` attributes with integers for both Hermetic and Opaque. For Hermetic, we consider the mode without differential privacy padding (HMT I) and the differential padding mode with ($\epsilon = 10^{-3}, \delta = 10^{-5}$) (HMT II). For Opaque, we consider the encryption (`enc`) and oblivious (`obl`) modes, and the oblivious memory size is fixed as 36MB. As the "oblivious pad mode" described in [99] is not implement in the release Opaque version, we estimate its performance using Hermetic's Full-Pad mode on BDB3. We also ensure that Hermetic and Opaque use the same query plans for all three queries.

The comparison results are shown in Figure 5. First, without differentially-private padding, Hermetic achieves comparable efficiency to Opaque in oblivious mode. The main reason is that both Hermetic and Opaque leverage an unobservable memory to accelerate the oblivious sort, and Hermetic realizes such memory while Opaque simply assumes it. The difference in BDB2 and BDB3 is caused by the different implementations: Opaque requires two oblivious sorts for both `groupby` and `join` while Hermetic uses three and eleven oblivious sorts for `groupby` and `join`, respectively. Hermetic could be optimized using the `groupby` algorithm in Opaque with one fewer oblivious sort. The Opaque's `join`, although with fewer oblivious sorts, is restricted to primary-foreign key join, and Hermetic's `join` can handle arbitrary inner equi-join, e.g. Q6. Second, because the sensitivities for BDB are small (1 for BDB1-2 and 22 for BDB3), the overhead from differential privacy padding in HMT II is very small compared to HMT I. Third, Full-Pad is 43x slower than HMT II due to huge padding overheads. In summary, even with stronger security guarantees, Hermetic achieves comparable efficiency to Opaque in oblivious mode. With comparable guarantees, Hermetic out-performs Opaque ("oblivious pad mode") substantially.

8.6 Trading-off privacy and performance

We tested the Hermetic query planner with weight vectors of various preferences to verify whether the planner could adjust the total privacy cost of all relations and the overall performance cost following the input weight vectors. We set the weight on privacy of each relation in Q_6 as identical, and increased the relative weight over performance cost. Figure 6 shows that the planner will sacrifice performance for lower privacy consumption when privacy is the first priority. The red dashed line indicates the performance cost of a plan without privacy awareness. Due to the unawareness, the plan has to assign the minimal privacy parameter to all the operators so as to handle the most private possible queries, and this could lead to huge inefficiency, e.g., 150x slow down, compared to Hermetic's plans for less private queries. Hermetic is also able to optimize the privacy cost on the individual relations based on the analyst's preferences, as expressed by the weight vector as shown in [93].

9 Related Work

Mitigating side channels: To our knowledge, Hermetic is the first practical system to offer rigorous protections against four side channels simultaneously. However, Hermetic builds on a substantial amount of prior work that inspired several of its components: for instance, Vuvuzela [88] and Xiao et al. [92] use differential privacy to obscure message sizes, Flicker [58] and SeCAGE [54] run protected code on locked-down cores, CaSE [97] executes sensitive code entirely in the cache, CATalyst [53] uses Intel's CAT against cache-based side channels, and TRESOR [62] protect AES encryption key from physical memory attacks by locking the secret key inside persistent CPU registers. None of these systems would work in our setting because they either cannot address all the four side channels or cannot support the large-scale federated data analysis. As we have tried to show, the devil is usually in the details; thus, building a *comprehensive* defense against several channels remains challenging.

It is well known that SGX, in particular, does not (and was not intended to [43]) handle most side channels [17], and recent work has already exploited several of them. These include side channels due to cache timing [14], BTB [51], and page faults [94]. Raccoon is a compiler that rewrites programs to eliminate data-dependent branches [77]. Its techniques inspire the code modifications that Hermetic uses to mitigate the IC and TC. Another challenge is the fact that IA-32's native floating-point instructions have data-dependent timing; Hermetic uses `libfixedtimefixedpoint` [4] to replace them. Some solutions aim to *detect* the channels [15, 16, 80] rather than block them, and their effectiveness highly depends on how the adversary attacks, and the adversary in our threat model can easily bypass their detection.

We emphasize again that, after the discovery of the Foreshadow attack [87], current SGX implementations are no longer secure. However, the underlying problem is with the implementation and not with the design, so Intel should be able to fix it in its next generation of CPUs. Hermetic assumes a correct implementation of SGX – an assumption it shares with the entire literature on SGX-based systems.

Oblivious RAMs [27, 61, 82, 83] can eliminate leakage through MC in arbitrary programs, but they suffer from poor performance in practice [77]. Moreover, ORAMs only hide the addresses being accessed, not the number of accesses, which could itself leak information [99].

Hermetic cannot fully mitigate physical side channels, such as power analysis [48] or electromagnetic emanations [49] because the underlying CPU does not guarantee that its instructions are data-oblivious with respect to them. (Intel claims that AES-NI is resilient to digital side-channels, but does not mention others [32].) However, these channels are most often exploited to infer a program's instruction trace, so, by making the IC of a query data-oblivious, Hermetic likely reduces these effectiveness of these channels.

Analytics on encrypted data: In principle, privacy-preserving analytics could be achieved with fully homomorphic encryption [25] or secure multi-party computation [10], but these techniques are still orders of magnitude too slow to be practical [10, 26]. 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 [71, 72]. In addition, some of these systems [28, 59, 76] have been shown to leak considerable information [20, 29, 64].

Alternatively, several systems [5, 8, 79] 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: M2R [19] and Ohrimenko et al. [68] 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. [6] introduce a set of data-oblivious algorithms for relational operators, based on oblivious sort. Ohrimenko et al. [69] extend this set with oblivious machine learning algorithms. Hermetic enhances these algorithms by making them resistant to IC, TC, and OC leakage and by speeding them up significantly using an OEE.

Opaque [99] is similar to Hermetic in that it combines TEEs, oblivious relational operators, and a query planner, and in that it shows substantial performance gains when small data-dependent computations are performed in oblivious execution environment. The key differences to Hermetic are: 1) Opaque does not mitigate the IC or TC, and it mitigates the OC by padding up to a public upper bound, which

may be difficult to choose; and 2) by assuming (!) that the 8MB L3 cache of a Xeon X3 is oblivious, Opaque's implementation effectively assumes the existence of an OEE, but does not show how to concretely realize one. Hermetic's OEE primitive would be one way for Opaque to satisfy this requirement, and it would also add protections against two additional side channels.

There are systems [1, 12, 46] that combine TEEs and differential privacy for privacy-preserving data analysis. Prochlo [12] is focused on side channels during data collection, and could benefit from our techniques for protection during data analysis. The past year has seen the first work [1, 46] that investigates using differential privacy to compute padding for the OC in a data analytics system. This work proposes algorithms for several query operators, such as range queries, but its leaves supporting a more functional subset of SQL that includes joins to future work. It also does not address how to integrate the privacy budget into query planning.

Query planning: There is some prior work on query planning with differential privacy; for instance, Pioneer [73] finds query plans that minimize ε and reuses earlier results when possible. However, we are not aware of any other work that jointly considers budget and performance, or that can trade one for the other, as Hermetic does with the leakage queries.

Workshop paper: We previously sketched Hermetic in a workshop paper [93]. The present paper improves on [93] by i) describing mitigations for all four side channels, instead of just one; and ii) presenting the results of a comprehensive evaluation.

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, new query operators that leverage differential privacy, and a novel privacy-aware query planner. Our experimental evaluation shows that Hermetic is competitive with previous privacy-preserving systems, even though it provides stronger privacy guarantees.

References

[1] J. Allen, B. Ding, J. Kulkarni, H. Nori, O. Ohrimenko, and S. Yekhanin. An algorithmic framework for differentially private data analysis on trusted processors. *CoRR*, 2018.

[2] AMPlab, University of California, Berkeley. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[3] Anati, Ittai and Gueron, Shay and Johnson, Simon P. and Scarlata, Vincent R. Innovative Technology for CPU Based Attestation and Sealing (March, 2017). https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[4] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *Proc. IEEE Symp. on Security & Privacy*, 2015.

[5] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *Proc. CIDR*, 2013.

[6] R. K. Arvind Arasu. Oblivious query processing. In *Proc. ICDT*, 2014.

[7] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. CCS*, Oct. 2010.

[8] S. Bajaj and R. Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. In *Proc. SIGMOD*, 2011.

[9] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS '68 (Spring)*, 1968.

[10] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. Smcql: Secure querying for federated databases. *Proc. VLDB Endowment*, 10(6):673–684, Feb. 2017.

[11] J. Beekman. Improving cloud security using secure enclaves. Technical Report No. UCB/EECS-2016-219, University of California, Berkeley, 2016.

[12] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proc. SOSP*, 2017.

[13] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. Technical Report arXiv:1701.00981, arXiv, 2017.

[14] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: Sgx cache attacks are practical. Technical Report arXiv:1702.07521, arXiv, 2017.

[15] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014.

[16] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *Proc. ASIACCS*, 2017.

[17] V. Costan and S. Devadas. Intel sgx explained. Technical Report Report 2016/086, Cryptology ePrint Archive, 2016.

[18] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proc. 25th USENIX Security*, 2016.

[19] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proc. USENIX Security*, 2015.

[20] F. B. Durak, T. M. DuBuisson, and D. Cash. What else is revealed by order-revealing encryption? In *Proc. CCS*, 2016.

[21] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. *Proc. EUROCRYPT*, 2006.

[22] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, 2006.

[23] C. Dwork, G. N. Rothblum, and S. Vadhan. Boosting and differential privacy. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 51–60. IEEE, 2010.

[24] S. Eskandarian and M. Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, 2017.

[25] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. STOC*, 2009.

[26] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. Technical Report Report 2012/099, Cryptology ePrint Archive, 2012.

[27] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[28] Google. Encrypted bigquery client, 2016.

[29] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proc. CCS*, 2017. To appear.

[30] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proc. CCS*,

2016.

[31] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *Proc. DIMVA*, 2016.

[32] Gueron, Shay. Intel Advanced Encryption Standard (IntelÂő AES) Instructions Set (March, 2017). https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set.

[33] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-Based WCET Estimation and Validation. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, 2009.

[34] B. Harangsri. *Query result size estimation techniques in database systems.* PhD thesis, The University of New South Wales, 1998.

[35] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *Proc. CSF)*, July 2014.

[36] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proc. IEEE Symp. on Security & Privacy*, May 1991.

[37] Intel. Combined Volume Set of IntelÂő 64 and IA-32 Architectures Software Developers Manuals (May, 2018). https://software.intel.com/en-us/articles/intel-sdm.

[38] Intel Corporation. Intel Software Guard Extensions SDK, Developer Reference (April, 2017). https://software.intel.com/en-us/sgx-sdk/documentation.

[39] Intel Corporation. Intel Trusted Execution Technology: White Paper (April, 2017). http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html.

[40] Intel Corporation. IntelÂő 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[41] Intel Corporation. Speculative execution side channel mitigations, revision 2.0, May 2018. Available at https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[42] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *Proc. VLDB*, 1992.

[43] S. Johnson. Intel SGX and side channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, Mar. 2017.

[44] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 1376–1385, 2015.

[45] M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network pump. *IEEE TSE*, 22:329–338, May 1996.

[46] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Accessing data while preserving privacy. *CoRR*, 2017.

[47] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[48] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. CRYPTO*, 1999.

[49] M. G. Kuhn. *Compromising emanations: eavesdropping risks of computer displays.* PhD thesis, University of Cambridge, 2002.

[50] B. W. Lampson. A note on the confinement problem. *CACM*, 16:613–615, Oct. 1973.

[51] 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. Usenix Security*, Aug. 2017.

[52] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[53] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proc. HPCA*, 2016.

[54] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proc. CCS*, 2015.

[55] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, 2005.

[56] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. Rote: Rollback protection for trusted execution. Technical Report Report 2017/048, Cryptology ePrint Archive, 2017.

[57] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proc. IEEE Symp. on Security & Privacy*, 2010.

[58] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. EuroSys*, 2008.

[59] Microsoft. Sql server 2016 always encrypted (database engine), 2016.

[60] R. Misener and C. A. Floudas. Piecewise-linear approximations of multidimensional functions. *J. Optim. Theory Appl.*, 145(1):120–147, 2010.

[61] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *Proc. S&P*, 2018.

[62] T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *Proc. USENIX Security*, 2011.

[63] J. Murtagh and S. Vadhan. The complexity of computing the optimal composition of differential privacy. In *Theory of Cryptography*, pages 157–175. Springer, 2016.

[64] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proc. CCS*, 2015.

[65] Nguyen, Khang T. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[66] Nguyen, Khang T. Usage Models for Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/cache-allocation-technology-usage-models.

[67] NYC Taxi & Limousine Commission. TLC Trip Record Data (April, 2017). http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[68] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *Proc. CCS*, 2015.

[69] 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.

[70] G. Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[71] 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.

[72] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Proc. IEEE Symp. on Security & Privacy*, 2014.

[73] S. Peng, Y. Yang, Z. Zhang, M. Winslett, and Y. Yu. Query optimization for differentially private data management systems. In *Proc. ICDE*, 2013.

[74] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *Proc. USENIX Security*, 2016.

[75] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Record*, 14(2):256–276, 1984.

[76] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing.

In *Proc. SOSP*, 2011.

[77] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proc. USENIX Security*, 2015.

[78] R. M. Rogers, A. Roth, J. Ullman, and S. P. Vadhan. Privacy odometers and filters: Pay-as-you-go composition. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 1921–1929. Curran Associates, Inc., 2016.

[79] 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. IEEE Symp. on Security & Privacy*, 2015.

[80] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proc. NDSS*, 2017.

[81] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In *Proc. CHES*, 2002.

[82] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *Proc. CCS*, 2013.

[83] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proc. CCS*, 2013.

[84] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *Proc. USENIX Security*, Aug. 2016.

[85] S. Tople and P. Saxena. On the trade-offs in oblivious execution techniques. In *Proc. DIMVA*, 2017.

[86] I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In *Proc. SIGMOD*, June 2014.

[87] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Varom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. USENIX Security*, 2018.

[88] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic attacks. In *Proc. SOSP*, 2015.

[89] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proc. IEEE Symp. on Security & Privacy*, 2013.

[90] V. Viswanathan. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[91] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *Proc. ESORICS*, 2016.

[92] Q. Xiao, M. K. Reiter, and Y. Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *Proc. CCS*, 2015.

[93] M. Xu, A. Papadimitriou, A. Feldman, and A. Haeberlen. Using differential privacy to efficiently mitigate side channels in distributed analytics. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec '18)*, Apr. 2018.

[94] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. IEEE Symp. on Security & Privacy*, 2015.

[95] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proc. USENIX Security*, 2014.

[96] 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.

[97] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. CaSE: Cache-assisted secure execution on ARM processors. In *Proc. Oakland*, 2016.

[98] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proc. CCS*, 2012.

[99] 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.

A Query operators

A.1 List of query operators

Query processing in Hermetic is built upon a set of oblivious operators. From the bottom up, we have *OEE operators* that support simple sort and merge, *auxiliary operators* that support relation transformations, including re-ordering, grouping, expansion, etc. and statistics, *relational operators* that support well-known SQL-style query processing. We describe all the operators below, together with their definitions and oblivious constructions. We also mark the one unique in Hermetic as `HMT`, and other from previous work with citations:

OEE Operators

`HMT merge-sort` Given an array of tuples that fits inside OEE and the set of order-by attributes, sort them in the order of the given attributes.

`HMT linear-merge` Given two arrays of sorted tuples, that together fit inside OEE and the set of order-by attributes, merge them into one sorted array of tuples with one linear scan.

Auxiliary Operators

`HMT hybrid-sort` Given an array of tuples, beyond the capacity of OEE, and a set of order-by attributes, sort them in the order of the given attributes. Algorithm 4 shows the pseudo-code of `hybrid-sort` (Lines 10-17).

`HMT hybrid-merge` Given two arrays of sorted tuples, that together are beyond the capacity of OEE and the set of order-by attributes, merge them into one sorted array of tuples. Algorithm 4 shows the pseudo-code of `hybrid-merge` (Lines 18-31).

[6] `augment` Given a relation, a function and an attribute name, returns a relation whose attributes consist of the given relation's attributes and the given attribute, and each row of which is the corresponding row in the given relation extended with the value of the function applied on the that row.

[6] `filter` Given a relation and a predicate on a set of attributes, return a relation that consists of all the rows from the given relation that satisfy the predicate. Algorithm 4 shows how to construct `filter` using `hybrid-sort` (Lines 1-9): first, augment the relation with "mark" equal to 1 if the row satisfy the predicate or 0 otherwise. Second, `hybrid-sort` the relation on "mark" to in descending order so that all the rows with "mark" equal to 1 are at the front. Finally, return all the rows that satisfy the predicate at the front.

[6] `groupId` Given a relation, a set of aggregate-on attributes, group the rows based on the aggregate-on attributes, and `augment` the relation with the group internal ID, starting from 1. The `groupId` operator could be constructed by first sorting the relation on the aggregate-on attributes using

hybrid-sort, and then keeping a running counter as the group internal ID while scanning over the sorted relation and extending each row with the group internal ID.

[6] group-running-sum Given a relation, a set of aggregate-on attributes, a summation attribute and a new attribute, augment the relation with the new attribute whose value is the running sum over the summation attribute, in the reverse order of the group internal ID, within each group of the aggregate-on attributes. The group-running-sum operator could be constructed by first applying groupId, then sorting the relation on the aggregate-on attributes, plus the group ID attribute in descending order, using hybrid-sort, and finally keeping a running sum over the summation attribute while scanning over the sorted relation and extending each row with the running sum.

[6] semijoin-aggregation Given two relations and a set of join attributes, augment each of the two relations with the number of matches, i.e. join degree, on the given set of join attributes from the other relation. To augment the first relation with the join degree, first augment each relation with attribute "src", e.g. 1(0) for the first(second) relation, union the two relations, and then hybrid-sort it on the join attributes plus the "src" attribute in ascending order. Then augment the relation with running counter of the rows in each group of the join attributes, but only increment the counter if the "src" is equal to 0. Finally, apply filter to keep rows with "src" equal to 1. Then, repeat on the second relation by switching the order.

[6, 69] expand Given one relation from the semijoin-aggregation output, duplicate each row by the times, equal to the number of matches, in the order of the join attributes. In Hermetic, this operation will leak the differentially private total join size.

[6] stitch Given two relations from expand, return a relation whose i-th row is the concatenation of the i-th rows from the two given relations.

HMT histogram Given a relation and a target attribute, returns a histogram over the given attribute of the relation. In Hermetic, we first hybrid-sort the relation on the target attribute, and then keep a running counter, refreshed to 0 at the beginning of each histogram bucket, and augment the relation with the running counter if the row is the last of the bucket, or -1 otherwise. Finally, hybrid-sort the relation on the augmented attribute in descending order, and return the rows with non-negative values.

HMT multiplicity Given a relation and a set of target attributes, calculates the number of times that the most common values of the target attributes appear in the relation. In Hermetic, we first hybrid-sort the relation on the set of target attributes, and then keep a counter on the most common values while scanning over the relation.

Relational operators

- project Given a relation and a set of attributes, returns a relation, each row of which is a projection, onto the given attributes, of the corresponding row in the input relation.
- rename Given a relation and a set of old and new names, returns a relation, whose specified attributes are renamed from old to new given names.
- union Given two relations, returns a relation whose attribute set is the union of the two relations and that contains all the rows from the two relations, with new attributes filled by null.

[6] select Given a relation and a predicate on a set of attributes, return a relation that consists of all the rows from the given relation that satisfy the predicate. The select operator could be constructed with one filter directly.

[6] groupby Given a relation, a set of aggregate-on attributes, an accumulate function and a new accumulate attribute name, group the rows in the given relation by the aggregate-on attributes, apply the accumulate function on each group and returns the aggregate-on attributes and the accumulation of each group. The groupby operator could be constructed with one groupId, followed by a group-running-sum, and, finally, a filter to keep the aggregate results only.

- orderby Given a relation and a set of order-by attributes, order the rows in the input relation by the order-by attributes. The orderby operator could be constructed with one hybrid-sort directly.
- cartesian-product Given two relations, returns a relation that consists of the concatenations of every pair of rows, one from the first relation and the other from the second relation.

[6] join Given two relations and a set of join attributes, returns a relation with every pair of matches from the two relations on the join attributes. The join operator could be constructed by first deriving the join degree of each row using semijoin-aggregation, then expanding each row by its join degree using expand and finally joining the two expanded relations using sticth to get the join result.

Algorithm 4: The oblivious filter operator

```
1 func filter(R = {t_0, t_1, ..., t_n}, p)
2     osize ← 0
3     foreach t ∈ R do
4         cwrite(p(t), match, 1, 0)
5         t ← t ∪ {('mark', match)}
6         osize ← osize + match
7     hybrid − sort(R, 'mark', desc = 0)
8     return R[0 : osize]
```

Table 4. Every x86 instruction used in Hermetic OEE

add	addl	cmovbe	cmove
cmovg	cmovle	cmovne	cmp
cmpl	imul	ja	jae
jb	jbe	je	jmp
jne	lea	mov	movl
movzbl	pop	push	ret
setae	setb	setbe	sete
setg	setle	setne	shl
shll	sub	test	

B Predictable timing for OEE operators

Hermetic currently performs two operations within OEEs, `merge-sort` and `linear-merge`. As discussed in Section 7.2, although OEEs allow these operators to perform data-dependent memory accesses (which is why they are faster than purely data-oblivious operators), they are carefully structured to avoid data-dependent control flow and instructions and to constrain the set of possible memory accesses. The latter makes the number of accesses that miss the L1 cache, and thus must be served by slower caches, predictable. Moreover, their execution time is padded to a conservative upper bound calibrated to a specific model of CPU. This section describes these measures in more detail.

B.1 Avoiding data-dependent control flow and instructions

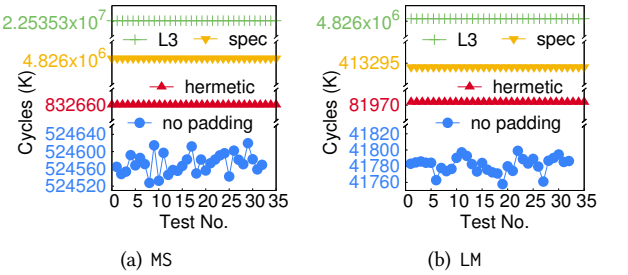
We ensure that OEE operators' code is free from data-dependent branches using techniques similar to [77]. Furthermore, we limit the set of instructions that they use to avoid those with data-dependent timing. Table 4 lists all of the x86 instructions that are used by the operators that run in an OEE. The instructions marked in dark green have constant execution time, as verified by Andrysco et al. [4]. The instructions in light green were not among the instructions verified by Andrysco et al., but are either variants of them or, as is the case with `cmov*`, are known to be constant time [77].

B.2 Making L1 cache misses predictable

By construction, all memory accesses performed by OEE operators are served from the cache. Moreover, they have deterministic control flow, and therefore perform a fixed number of memory accesses for a given input size. For example, Algorithm 5 shows the pseudocode for OEE `merge-sort`. Note that the two running pointers that scan over the two sublists will keep accessing the data even if one of the sublists has been completely merged (Lines 9 and 10). This will not affect the correctness due to the modified merge condition (Line 11), but it will make the total number of memory accesses on each input deterministic. Nevertheless, the operators' timing could vary depending on whether accesses hit the L1 cache or whether they have to be served by slower caches.

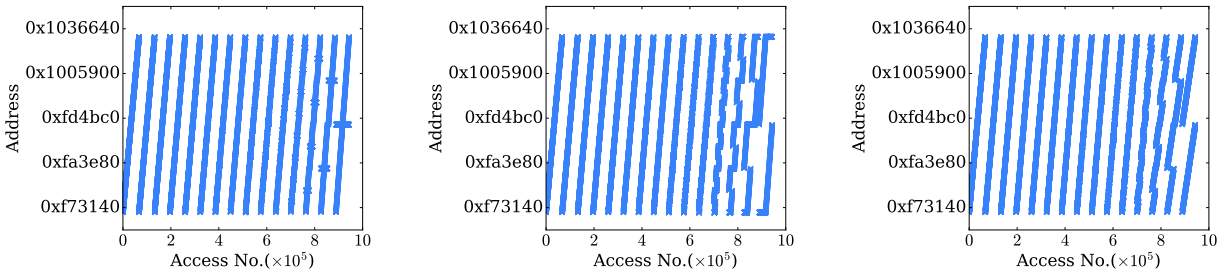
Algorithm 5: The `merge-sort` supported in OEE.

```
1  func merge − sort({R = {t_0, t_1, ..., t_n} :
     B}, REAL,'', ascend)
2    for len ∈ {2^0, 2^1, ..., 2^log2(n)} do
3      for offset ∈ {0, 2·len, ..., n − 2·len} do
4        pos_0 ← offset;
5        pos_1 ← offset + len;
6        mov eax, len; add eax, eax
7        mov ebx, pos_0; mov ecx, pos_1
8        lea edx, [ebx]; lea edi, [ecx]
9        LOOP: cmp edx, edi
10       cmovle esi, edx; cmovg esi, edi
         /* Merge [pos_0] if pos_1 ≥ 2·len      */
11       mov esi, len; mov esi, esi
12       cmp esi, ecx; cmovle esi, edx
         /* Merge [pos_1] if pos_0 ≥ len        */
13       mov esi,len; cmp esi, ebx
14       cmovle esi, edi
15       mov [-eax], esi
         /* update pos_0                        */
16       mov esi, $0; cmp edx, edi
17       cmovle esi,$1; add ebx, esi
18       mov esi, len; cmp ebx, esi
19       cmovg ebx, esi
         /* update pos_1                        */
20       mov esi, $0; cmp edx, edi
21       cmovg esi, $1; add ecx, esi
22       mov esi, len; add esi,esi
23       cmp ecx, esi; cmovg ecx, esi
         /* load the next item                  */
24       cmp edx, edi; cmovle esi, ebx
25       cmovg esi, ecx; lea esi, [esi]
26       cmovle edx, esi; cmovg edi,esi
         /* decrement the counter               */
27       sub eax,$1; cmp eax,$0;ja LOOP
28       R[offset : offset + 2·len] ← B[offset :
           offset + 2·len];
```



(a) MS (b) LM

Figure 7. Cycle-resolution measurements of the actual timing of `merge-sort` (MS) and `linear-merge` (LM) inside the OEE, and their padded timing, respectively.



(a) Sorted (b) Reverse (c) Random

Figure 8. Memory access patterns of OEE `merge-sort` on sorted, reverse sorted and random input.

To address this problem, we could determine an upper bound on an operator's execution time by assuming that all of its memory accesses are served from the L3 cache, but this would be wildly conservative. In particular, it would result in a 43x slowdown for OEE `merge-sort` and a 33x slowdown for OEE `linear-merge` (see Figure 7). Instead, we carefully structure each operator's code to make the number of L1 cache misses predictable. For example, if we examine the code of `merge-sort`, we can see that its memory accesses adhere to three invariants:

1. *Each merge iteration accesses one of the same tuples as the previous iteration.* Figure 8 shows the memory traces of `merge-sort` on 32,768 input tuples that have been sorted, reverse sorted, and permuted randomly. With sorted input, the two running pointers (Lines 9 and 10 in Algorithm 5) follow the invariant: the second pointer keeps accessing the first tuple of the second sub-list until the first pointer finishes scanning through the first sub-list. Then, the first pointer keeps accessing the last tuple of the first sub-list until the second pointer reaches the end of the second sub-list. The same memory access pattern holds even with reverse sorted and randomly permuted inputs because only one of the two running pointers would advance after each merge iteration. If we assume that a tuple can fit in a single L1 cache line, as is the case in our examples, then one of the two memory accesses in each iteration will be an L1 hit.
2. *Merge iterations access the input tuples sequentially.* The merge loop (Lines 6–15) accesses the tuples in each of the input sub-lists sequentially. Consequently, if each tuple is smaller than an L1 cache line, then accessing an initial tuple will cause subsequent tuples to be loaded into the same cache line. Assuming that the L1 cache on the OEE's CPU is large enough – as is the case in our experiments – these subsequent tuples will not be evicted from the cache between merge loop iterations, and future accesses to them will be L1 hits. If a cache line is of CL bytes and each tuple is of TP bytes, then at least $1 - \frac{TP}{CL}$ of the tuple accesses will be L1 hits.
3. *Local variables are accessed on every iteration.* Since `merge-sort` is not recursive, the variables local to the merge loop (Lines 7–14) are accessed on every iteration. Furthermore, because more cache lines are allocated to the OEE than local variables and tuples, these accesses should be L1 hits as well.

Given these invariants, it is possible to express the lower bound on L1 hits as a formula. Let N be the number of tuples per OEE input block, FPR as the number of fields in each tuple, and FNO as the number of fields used as keys for sorting. Then, the lower bound is given by:

$$\begin{aligned} L_{ms} &= (77 + 11 * FNO + 12 * FPR) * N * log(N) \\ &+ (1 - \frac{FPR}{16})(12 + FPR) * N * log(N) \\ &+ 5N * log(N) + \frac{15}{4} N * log(N) \end{aligned} \tag{1}$$

A similar analysis can be done for `linear-merge`, resulting in the following lower bound formula:

$$\begin{aligned} L_{lm} &= (69 + 24 * FNO + 11 * FPR) * N + 14 \\ &+ (1 - \frac{FPR}{16})(12 + FNO + FPR) * N \\ &+ \frac{15}{4} N \end{aligned} \tag{2}$$

Plugging in the values for N, FPR, FNO from our experimental data, we can see that the majority of accesses are served by the L1 — approximately 89.06% and 79.73% for `merge-sort` and `linear-merge`, respectively.

B.3 Determining a conservative upper bound on execution time

Even though the number of L1 cache misses is predictable regardless of the input, as discussed in Section 7.2, we still pad OEE operators' execution time to a conservative upper bound to account for timing variation that might occur in

Table 5. 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.

Data	l_{L1}	l_{L3}	l^*_{L1}	l^*_{L3}	$\hat{l}_{L1}$	$\hat{l}_{L3}$
Random	4	34	0.68	3.34	0.74	5.0
Ordered			0.6693	3.8032		
Reverse			0.6664	4.263		

modern CPUs (e.g., due to pipeline bubbles).[5] To determine this upper bound, we could take the lower bound on L1 hits determined above and assume that all other memory accesses were served from the L3 cache (LLC). We could then compute the bound by plugging in the L1 and L3 access latencies from processor manual [40]. As Figure 7 shows, however, due to the superscalar execution in modern CPUs, the resulting bound is still 10x larger than the actual execution time.

Instead, we achieve a tighter but still conservative bound using worst-case execution time (WCET) estimation techniques [33]. We performed 32 experiments each on random, sorted, and reverse sorted inputs in which we measured the L1 hit and miss rates using the CPU's performance counters. We then used linear regression to learn *effective* L1 and L3 hit latencies l^*_{L1} and l^*_{L3} for the specific CPU model. Since we could not be sure that we have observed the worst case in our experiments, we increased the L1 and L3 latency estimates by 10% and 20%, respectively to obtain bounds $\hat{l}_{L1}$ and $\hat{l}_{L3}$. Table 5 shows l^*_{L1}, l^*_{L3}, $\hat{l}_{L1}$, and $\hat{l}_{L3}$ estimated for `merge-sort`, as well as the latencies from the specification. The computed upper bounds were 1.6x the actual execution time for `merge-sort` and 1.96x for `linear-merge` and were never exceeded in our experiments.

The effective L1 and L3 hit latencies would have to be derived on each distinct CPU model. To do so, we envision a profiling stage that would replicate the procedure above and would be performed before Hermetic is deployed to a new processor.

B.4 Overhead of time padding

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 9. The overhead of time padding ranges between

[5]We determine execution time using the `rdtsc` instruction. `rdtsc` is available to enclaves in SGX version 2 [37]. Moreover, a malicious platform cannot tamper with the timestamp register because the core is "locked down."

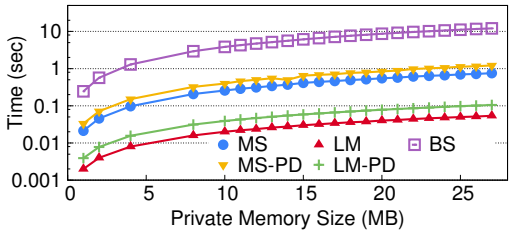


Figure 9. Latency of `merge-sort` (`MS`), `linear-merge` (`LM`) with increasing un-observable memory size, compared to the `batcher-sort` (`BS`). `-PD` indicates time padding.

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 DOA-NoOEE , even though Hermetic provides stronger guarantees.

C Oblivious primitives which use dummy tuples

Section 5 mentions that we modified the oblivious primitives from prior work [6] to accept dummy tuples. These modifications have two goals: (1) allowing the primitives to compute the correct result on relations that have dummy tuples, and (2) providing an oblivious method of adding a controlled number of dummy tuples to the output of certain primitives.

C.1 Supporting dummy tuples

Dummy tuples 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.

`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 tuples, we need to make sure that dummy records do not get grouped with real tuples. To avoid this, we expand the set of grouping attributes by adding the `isDummy` attribute. The result is that real tuples 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 tuples 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 dummy tuples, 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 dummy tuples, 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 dummy tuples are selected.

join: What we want for join is that real tuples from the one relation are joined only with real tuples 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 tuple of one relation with every tuple of the other, and this happens even for dummy tuples. 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 tuples are real and 0 otherwise.

multiplicity and histogram: These two primitives need to return the corresponding statistics of the real tuples. Therefore, we make sure to (obliviously) exclude dummy tuples for the computation of multiplicities and histograms.

C.2 Adding dummy tuples to the primitive outputs

To enable the introduction of dummy tuples, we alter the primitives filter, groupby, and join. The oblivious filter primitive from previous work 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 tuples which do not satisfy the predicate. To obliviously add N tuples, we keep N of the previously discarded tuples, making sure to mark them as dummy.

groupby queries involve several stages, but their last step is selection. Therefore, dummy tuples can be added in the same way.

join queries involve computing the join-degree of each tuples in the two relations.[6] To add noise, we modify the value of join-degree: instead of the correct value, we set the join-degree of all dummy tuples to zero, except one, whose degree is set to N. As a result, all previous dummy tuples are eliminated and N new ones are created.

D Hermetic multi-objective query optimization

Hermetic's query planner uses multi-objective optimization [86] to find the optimal query plan that matches the user's priorities. A query plan is associated with multiple costs, including the overall performance cost and a vector of privacy costs across the involved relations. The user's specification includes a vector of bounds, $\mathbf{B}$, and a vector of weights, $\mathbf{W}$, for the privacy costs on all the input relations. The planner's output is the plan where the weighted sum of all the costs is as close to optimal as possible and where all of the privacy costs are within the bounds. Each plan that the planner considers could be represented as a join tree covering all the input relations, with each noised operator assigned a privacy parameter, ε_i. (The current query planner only considers different join orders and privacy parameters. We leave more advanced query optimization to future work.)

The planner first constructs the complete set of alternative plans joining the given set of relations. Then, for each of the candidate plans, the planner formalizes an optimization problem on the privacy parameters of all the noised operators, and solves it using linear programming. Finally, the plan that both meets the bounds and has the best weighted-sum of costs is selected.

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:

$$\begin{aligned} &\min \mathbf{W} \cdot \mathbf{A} \cdot \boldsymbol{\varepsilon} + f_p(\boldsymbol{\varepsilon}) \\ &s.t.\ \mathbf{A} \cdot \boldsymbol{\varepsilon} \leq \mathbf{B}, \mathbf{0} < \boldsymbol{\varepsilon} \leq \mathbf{B}. \end{aligned} \tag{3}$$

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:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{4}$$

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. As a result, Hermetic approximates the solution instead. The planner first partitions the entire parameter space into small polytopes, and then approximates the performance cost function on each polytope as the linear interpolation of the vertices of the polytope. Thus, it achieves a piecewise linear approximation of the performance cost function. Finally, the planner can solve the corresponding piecewise linear programming problem to obtain an approximately-optimal assignment for the privacy parameters. This approach is consistent with existing nonlinear multidimensional optimization techniques in the optimization literature [60].

Let $f_p(\boldsymbol{\varepsilon})$ be the performance cost function, d be the number of operators in the query plan, and K be the number of partitions on each parameter dimension. Then, the entire parameter space could be partitioned into K^d polytopes, each

[6] In a join between relations R and S, the join-degree of a tuple in R corresponds to the number of tuples in S whose join attribute value is the same with this row in R.

of which has 2^d vertices. We pick one of the vertices, $\boldsymbol{\varepsilon_0}$, and d other vertices, $\boldsymbol{\varepsilon_1}, \ldots, \boldsymbol{\varepsilon_d}$, each of which is different from $\boldsymbol{\varepsilon_0}$ in exactly one parameter dimension. Then any point, $\boldsymbol{\varepsilon}$, as well as its performance cost, $f_p(\boldsymbol{\varepsilon})$, in such polytope could be represented as:

$$\begin{aligned}(\boldsymbol{\varepsilon}, f_p(\boldsymbol{\varepsilon})) = &\sum_{i=1}^{d} u_i * ((\boldsymbol{\varepsilon_i}, f_p(\boldsymbol{\varepsilon_i})) - (\boldsymbol{\varepsilon_0}, f_p(\boldsymbol{\varepsilon_0}))) \\ &+ (\boldsymbol{\varepsilon_0}, f_p(\boldsymbol{\varepsilon_0})) \\ \text{, where } &0 \le u_i \le 1, \\ &0 \le u_i + u_j \le 2, \\ &0 \le u_i + u_j + u_k \le 3, \\ &\ldots \\ &0 \le \sum_{i=1}^{d} u_i \le d.\end{aligned} \tag{5}$$

For each such polytope, we can plug Equations 5 into Equation 3 to obtain the following linear programming problem:

$$\begin{aligned}\min\ &\mathbf{W} \cdot \mathbf{A} \cdot \boldsymbol{\varepsilon} + \\ &\sum_{i=1}^{d} u_i * (f_p(\boldsymbol{\varepsilon_i}) - f_p(\boldsymbol{\varepsilon_0})) + f_p(\boldsymbol{\varepsilon_0}) \\ s.t.\ &\mathbf{A} \cdot \boldsymbol{\varepsilon} \le \mathbf{B}, \\ &\mathbf{0} < \boldsymbol{\varepsilon} \le \mathbf{B}, \\ &\boldsymbol{\varepsilon} - \sum_{i=1}^{d} u_i * (\boldsymbol{\varepsilon_i} - \boldsymbol{\varepsilon_0}) = \boldsymbol{\varepsilon_0}, \\ &0 \le u_i \le 1, \\ &0 \le u_i + u_j \le 2, \\ &0 \le u_i + u_j + u_k \le 3, \\ &\ldots \\ &0 \le \sum_{i=1}^{d} u_i \le d.\end{aligned} \tag{6}$$

Solving this linear programming problem on all the polytopes enables the query planner to determine the optimal assignment of privacy parameters for the plan.

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 [42] 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.