Timing Attacks on Differential Privacy are Practical

Zachary Ratliff
zacharyratliff@g.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

Nicolas Berrios*
berrios@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

James Mickens
mickens@g.harvard.edu
Harvard University
Cambridge, Massachusetts, USA

Abstract

Differential privacy (DP) has become a standard approach for computing privacy-preserving statistics. However, in interactive settings, the observable *runtime* of DP queries can inadvertently leak sensitive information, violating privacy guarantees. Prior work has shown that timing side channels can undermine DP in specific settings. In this work, we show that popular libraries for implementing differential privacy, including diffprivlib, OpenDP, and PyDP, frequently introduce such timing side channels, leading to measurable privacy degradation. Our analysis reveals timing vulnerabilities not only within commonly used DP mechanisms (e.g., private sums, counts, means, and selection) but also in commonly used pre-processing steps such as filtering and sorting. We show that these seemingly innocuous operations frequently exhibit runtimes that are sensitive not only to the presence of an individual's data in the input but also to the ordering of the input data.

Several of the discovered timing side channels arise from programs whose runtimes depend on the *size* of the input dataset. The distinction between whether the dataset size is considered private or public information corresponds to *bounded* versus *unbounded* DP. We show that mechanisms satisfying *unbounded* DP with respect to their output distributions often trivially reveal their input size through their runtime distributions. We give several examples of practical attacks that can be used to re-identify individuals in a dataset given such a timing side channel.

Finally, we propose an empirical auditing technique for detecting timing side-channel vulnerabilities in DP implementations. Our auditing algorithm provides a lower bound on privacy loss when both the program's output and runtime are observable to an adversary. Using our auditing framework, we are able to quantify conservative bounds on the privacy leakage of these mechanisms when runtimes are observable to an adversary.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Timing attacks, side channels, differential privacy

*Much of this work was completed while the author was at Harvard University.

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765146

ACM Reference Format:
Zachary Ratliff, Nicolas Berrios, and James Mickens. 2025. Timing Attacks on Differential Privacy are Practical. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3719027.3765146

1 Introduction

Differential privacy (DP) [11] is becoming more widely used for privacy-preserving data analysis. A differentially private mechanism has the property that its output is insensitive to the presence or absence of any single person's data in the input. This guarantee is often achieved by adding carefully tailored noise to the mechanism's output, ensuring that the output distribution is insensitive to whether a particular individual's data is included in the analysis.

Unfortunately, the strong theoretical guarantees of DP algorithms are often undermined by weaknesses in their *implementations* [4, 9, 14, 17, 22]. One such class of weaknesses are timing side channels that emerge from the execution time of the DP algorithm. For example, even if the algorithm achieves DP with respect to its output, the act of computing that output may involve data-dependent branches; the execution (or lack of execution) of conditional code paths will impact the algorithm's running time, potentially leaking information about the sensitive input to an observer. Exploiting this timing information is commonly known as a *timing attack* and has been used in several high-profile attacks on other domains, such as cryptography, over the past couple of decades [1, 6, 7, 18, 19, 27, 31].

A particularly egregious example of timing side channels is one discovered by Haeberlen, Pierce, and Narayan [14] on the PINQ DP database system [21]. PINQ supports arbitrary user-defined queries which can contain logic like the following:

```
def user_query(dataset):
    if "Bob" in dataset:
        sleep(1000)
```

The runtime of such a query trivially leaks whether the input dataset contains the targeted user, violating differential privacy. Restricting user-defined queries (and forcing the DP system to use constant-time algorithms for scanning the database) will eliminate some timing side channels, but others can remain. For instance, timing side channels also appear in the samplers used by many DP algorithms. This was highlighted by Jin, McMurtry, Rubinstein, and Ohrimenko [17], who showed that implementations of the Discrete Laplace and Discrete Gaussian samplers are vulnerable to side channel attacks. In particular, these attacks leveraged the fact that the runtime of the samplers is strongly correlated with the amplitudes of the outputted samples [17]. If the sampled value is used to noise a mechanism's output for privacy, an attacker can infer the sampled

value by analyzing the mechanism's runtime, and strip away the noise from the output to reveal the underlying private value.

Beyond the known issues with samplers and user-defined queries, the scope of timing side channels in DP implementations remains largely unexplored. Given that DP systems are intended to handle large datasets and execute numerous data-dependent operations, one may reasonably anticipate widespread timing vulnerabilities. Nevertheless, it remains unclear whether observing such timing variations leads to substantial degradation of privacy guarantees in practice.

1.1 Overview and Contributions

To this end, we systematically examine timing side-channel attacks across various programming frameworks for differential privacy, including diffprivlib [15], PyDP [25], and OpenDP [12, 30]. These libraries provide modular building blocks that implement core DP functionalities, such as chaining together dataset *transformations* (e.g., filtering rows according to some predicate, imputing dataset entries, computing sums and averages, etc.), as well as randomized *measurements* (e.g., adding Laplace or Gaussian noise) that provide DP guarantees on the program's output. We give more background on DP programming frameworks, as well as transformations and measurements, in Section 2.

We are able to demonstrate successful timing attacks across the various DP implementations. We targeted functionality frequently used in practical DP workflows, including dataset filtering, sorting operations, private partition selection, and releasing aggregate statistics such as sums and counts. Several of our attacks exploit DP mechanisms that are intended to provide privacy guarantees within the *unbounded* DP setting (§2), but leak their input sizes through their observable runtimes. The *bounded* DP setting assumes the dataset size is public, and thus such leakage would not be problematic. However, in the unbounded DP setting, dataset size is sensitive, and leaking it can create a domain mismatch: dataset transformations that are assumed to be chained with an unbounded DP mechanism effectively become chained with a bounded DP mechanism. To see why this poses a problem, consider the ε-DP mechanism illustrated in Figure 1. The mechanism uses a filtering transformation T_{Filt} to remove rows from the dataset that match a given criteria (e.g., rows where the age attribute is < 30). The filtering transformation is then *chained* with a summation transformation T_{Sum} that creates a non-private sum over some attribute of interest. Finally, the T_{Sum} transformation is chained with a DP Laplace measurement M_{Lap} to release a DP sum over the sensitive attribute for individuals in the dataset who are younger than 30. In this setting, it is critical that the program $M_{\text{Lap}} \circ T_{\text{Sum}}$ achieves DP with respect to the unbounded setting, otherwise the chained mechanism $M_{\text{Lap}} \circ T_{\text{Sum}} \circ T_{\text{Filt}}$ directly reveals the number of individuals in the dataset that are younger than 30 (since the input size to $M_{\text{Lap}} \circ T_{\text{Sum}}$ would be considered public).

The above vulnerability consistently arises throughout our analysis, and we give a practical attack on the described filtering mechanism in Section 4.4. This demonstrates that leaking just the dataset size through timing side channels can result in severe privacy attacks when those mechanisms are chained together to create more complex functionalities.

Another key observation is that many DP mechanisms assume their inputs are *unordered* multisets of items (e.g., datasets considered as unordered collections of elements from a row domain). However, the runtimes of many DP algorithms are often influenced by the ordering of their inputs. For instance, algorithms that perform an initial sort (see Section 4.1) can exhibit easily distinguishable runtime distributions on datasets that differ only in their ordering. We demonstrate this effect explicitly in the DP trimmed mean algorithm (§4.1), which first sorts its input dataset, and further show how such ordering-dependent timing leakages can be extended to enable membership inference attacks. This insight suggests that developers should consider ordered input metrics when reasoning about timing privacy. Using unordered dataset metrics has led to widespread under-estimation of global sensitivity in DP libraries [9], and such issues may also arise if one is not careful when reasoning about timing attack defenses.

While the existence of the discovered timing side channels is not entirely unexpected (since the libraries do not claim to prevent timing attacks[1]), what *is* surprising is the pervasiveness of these side channels throughout the DP libraries. As currently implemented, many DP mechanisms could be exploited by an adversary with access to their runtime measurements. In an interactive query system, obtaining these measurements could be as simple as recording the time between sending a request and receiving a response. To better assess the risk posed by timing side channels, we propose an auditing algorithm to analyze the joint distribution of a mechanism's output and runtime (§5). Our algorithm is an application of the general statistical auditing method proposed by Jagielski, Ullman, and Oprea [16], and establishes an empirical lower bound on the privacy parameters when both the output and runtime are observable. Using our auditing algorithm, we demonstrate that several of our timing attacks can lead to severe privacy degradation in practical settings. These findings suggest that DP libraries should prioritize timing attack mitigations before deploying DP in interactive settings.

We summarize our contributions as follows:

- We investigate timing side channels in the PyDP, diffprivlib, and OpenDP libraries, discovering timing vulnerabilities across a range of widely-used mechanisms (§4). We find that seemingly innocuous data operations—such as filtering, sorting, and multiplication by a constant—can introduce severe timing side channel vulnerabilities that are observable in real-world network environments.
- We observe that implementations of *unbounded* DP mechanisms often trivially reveal their input size through their runtime distributions. Unfortunately, in many settings, revealing the program's input size can be leveraged to perform more severe privacy attacks such as membership inference. In Section 6, we provide a detailed discussion on mitigating such attacks and examine the limitations of standard defenses, such as enforcing constant-time execution.
- We provide an auditing algorithm for assessing the risk of timing attacks on DP implementations (§5), and connect this auditing algorithm to recent definitional frameworks for jointly output/timing-private DP programs [5, 28].

[1] In fact, both OpenDP and PyDP explicitly indicate that they do not protect against timing attacks.

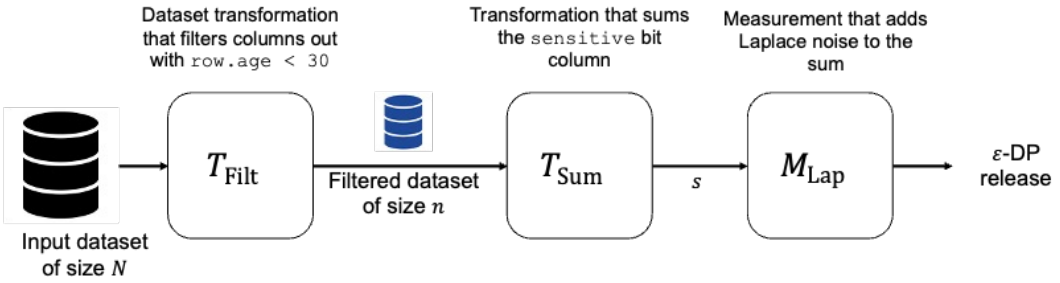

Figure 1: An example filtered DP summation program within the OpenDP programming framework. The program uses a filtering transformation $T_{\texttt{Filt}}$ to remove rows from the dataset whose age attribute is < 30 to produce a new filtered dataset. The filtered dataset is then passed as input via chaining to a T_{Sum} transformation that outputs the sum s over a sensitive bit column on the filtered dataset. Finally, the sum is passed as input to a Laplace measurement M_{Lap} that adds appropriately scaled noise to produce a ε-DP sum. We give an attack on such a mechanism in Section 4.4.

We emphasize that the objective of this paper is *not* to conduct a comprehensive audit of timing attacks in existing DP libraries. Rather, our goal is to broadly investigate the landscape of timing attacks on differential privacy implementations, extending beyond the well-known attacks on user-defined queries and sampling mechanisms. Additionally, we aim to enhance our understanding of how these timing attacks impact privacy guarantees in practice, an area that remains relatively underexplored.

2 Background

Throughout this paper, we let datasets be a vector $X = [x_1, x_2, \ldots, x_n]$ of elements from some data domain $\mathcal{D}$, i.e., $X \in \mathcal{X} = \cup_{n \geq 0} \mathcal{D}^n$. We typically consider each element (row) of the dataset to correspond to a single individual's data. We will use $|X|$ to denote the size of the dataset and define $h_X(z)$ to be the number of occurrences of the element z in dataset X.

2.1 Dataset Metrics

We now review common dataset distance metrics used by implementers of differential privacy.

Definition 1 (Change-One Adjacency). *We say that two equally-sized datasets X and X' are adjacent with respect to the change-one distance metric if*

$$\sum_{\substack{z \in \mathcal{D} \\ h_X(z) > h_{X'}(z)}} |h_X(z) - h_{X'}(z)| = 1$$

We use $d_{\text{CO}}(X, X')$ to indicate the change-one distance between datasets X and X'.

Definition 2 (Symmetric Adjacency). *We say that two datasets X and X' are adjacent with respect to the symmetric distance metric if*

$$\sum_{z \in \mathcal{D}} |h_X(z) - h_{X'}(z)| = 1$$

We use $d_{\text{Sym}}(X, X')$ to indicate the symmetric distance between datasets X and X'.

The choice of adjacency has implications for which aspects of the dataset are public. For example, the change-one notion of adjacency assumes that the dataset size is public; two equally-sized datasets X and X' are adjacent if they differ only by a single row. In contrast, the *symmetric* adjacency notion says that X can be transformed into X' by adding or removing a single row; thus, the symmetric distance metric implicitly assumes that the dataset size is private. When we use the change-one adjacency metric, we say that we are operating in the *bounded* model. When we use the symmetric adjacency metric, we say that we are operating in the *unbounded* model.

Both change-one and symmetric adjacency are *unordered* notions of adjacency. However, many algorithms have runtimes that are highly sensitive to the order of their input data. Thus, we also consider the ordered analogs of the change-one and symmetric adjacency metrics.

Definition 3 (Hamming Adjacency). *We say that two equally-sized datasets X and X' are adjacent with respect to the Hamming distance metric if there exists an index i such that $x_i \neq x'_i$ and $x_j = x'_j$ for all $j \neq i$.*

We use $d_{\text{Ham}}(X, X')$ to indicate the Hamming distance between datasets X and X'.

Definition 4 (Insert-Delete Adjacency). *We say that datasets $X = [x_1, \ldots, x_n]$ and $X' = [x'_1, \ldots, x'_{n+1}]$ are adjacent with respect to the insert-delete distance metric if X' can be obtained from X by inserting a single element x'_i at position i or if X can be obtained from X' by deleting a single element x'_i at position i.*

We use $d_{\text{ID}}(X, X')$ to indicate the insert-delete distance between datasets X and X'.

Both Hamming adjacency and change-one adjacency correspond to *bounded* DP, where the dataset size is considered public information. Conversely, both insert-delete adjacency and symmetric adjacency allow for datasets to differ by the presence or absence of an element, and therefore correspond to *unbounded* DP, where dataset size is considered private and must be protected by DP.

2.2 Differential Privacy

We now review the definition of differential privacy.

Definition 5 (Differential Privacy [11]). *A mechanism $M : \mathcal{X} \to \mathcal{Y}$ is (ε, δ)-differentially private if for all adjacent databases $X, X' \in \mathcal{X}$ and all $S \subseteq \mathcal{Y}$:*

$$\Pr[M(X) \in S] \leq e^{\varepsilon} \cdot \Pr[M(X') \in S] + \delta$$

When $\delta = 0$, we say that M achieves *pure* differential privacy. Intuitively, differential privacy guarantees that the algorithm's output distributions on adjacent datasets will be "close" (where "close" is determined by the parameters ε and δ). This gives privacy to individuals by ensuring that the algorithm's *output* is essentially the same regardless of whether their data is included. The definition, however, does not capture the fact that the algorithm's runtime may vary greatly on adjacent datasets.

To account for runtime, recent theoretical work has defined programs that are jointly output/timing DP (JOT-DP) [5, 28]. These definitions ask that the joint random variable of the program's output and runtime achieve differential privacy.

Definition 6 (Joint Output/Timing Privacy [5, 28]). *A program $P : \mathcal{X} \times \mathcal{E} \to \mathcal{Y} \times \mathcal{E}$ is (ε, δ)-JOT-DP if for all adjacent datasets $X, X' \in \mathcal{X}$, all input-compatible execution environments* $\mathsf{env}, \mathsf{env}'$*, and all $O \subseteq \mathcal{Y} \times \mathcal{T}$*

$$Pr[(Y, T) \in O] \leq e^{\varepsilon} \cdot \Pr[(Y', T') \in O] + \delta$$

where $Y = P(X, \mathsf{env})$, $Y' = P(X', \mathsf{env}')$, $T = T_P(X, \mathsf{env})$ and $T' = T_P(X', \mathsf{env}')$.

We use the notation $T_P(X, \mathsf{env})$ to indicate the runtime of program P on input X and in execution environment env. The above definition is effectively the standard definition of DP applied to the joint random variable of the program's output and runtime. Note, however, that the condition of DP must hold over *all* (input-compatible)[2] execution environments. This distinction is important since successive queries can alter the execution environment that introduce new timing side channels. We define the execution environment as encompassing every aspect of the computational setting that may affect the program's runtime on a given input. Specifically, in our setting, the execution environment includes concurrent processes running on the same system, network jitter and latency between the adversary issuing the query and the endpoint executing the query, and all relevant microarchitectural characteristics of the machine executing the program (e.g., processor cache states, speculative execution behaviors, branch predictors, memory hierarchy states). In practice, the complexity of the execution environment can vary considerably: it can be highly intricate, encompassing a large variety of microarchitectural details and nondeterministic concurrent behavior, or it can be substantially simplified by tightly controlling the computational environment, for example, by disabling speculative execution, using a minimalist instruction set architecture (such as a reduced subset of the x86 ISA), eliminating concurrent system processes, or running the program on dedicated and isolated hardware. We give a more detailed discussion on the role of the execution environment in (§5).

[2]The notion of input-compatibility is a technical detail in the privacy definition that accounts for the fact that some execution environments may be incompatible with a given set of inputs. For example, the execution environment corresponding to memory being zeroed out is incompatible with all datasets of size > 0.

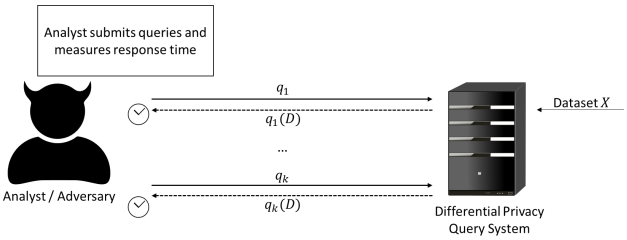

Figure 2: Threat model for timing attacks on DP systems. An adversary (analyst) adaptively issues queries q_i to the DP system. Each query is executed on the sensitive dataset, and the system returns a noisy output. In addition, the adversary can measure the total runtime of each query execution. We require that the joint distribution of output/runtime pairs satisfy joint output/timing privacy (Definition 6).

2.3 DP Programming Frameworks

Throughout this paper, we use terminology for DP programming frameworks, much of which was first introduced by McSherry [21], and further formalized by Gaboardi, Hay, and Vadhan [13]. In a DP programming framework, DP algorithms are constructed by *chaining* together sequences of **transformations** and **measurements**. A transformation is a mapping from datasets to datasets, such as filtering, clipping, or aggregation, which does not itself introduce randomness but potentially modifies the sensitivity of a given query. Measurements, on the other hand, introduce randomness to satisfy DP guarantees. Developers can then chain together transformations and measurements to construct more sophisticated DP algorithms while enabling modular reasoning about privacy guarantees.

3 Threat Model

The standard definition of differential privacy does not account for adversaries capable of observing the mechanism's runtime. In this work, we consider a threat model where both the output and runtime of the mechanism are accessible to the adversary (Figure 2). This threat model is relevant, for instance, in settings where an analyst submits queries to a DP system that then executes the queries on a sensitive dataset and returns the results [14, 21]. Additionally, the threat model allows for adaptive analysts who can select their queries based on the results of previous queries.

In each of our timing attacks (§4), an adversary issues DP queries to a trusted server. The adversary is able to specify the privacy budget expended on each query. We further assume that the adversary can construct arbitrary queries using the operations made available by the DP library. For example, since OpenDP exposes dataset transformations such as clamping, we allow the adversary to issue arbitrary queries that make use of these transformations, and they may apply them multiple times if they choose. The server executes the queries on a sensitive dataset and returns the result to the adversary, who measures the response time. Given a set of

outputs and corresponding response times, we ask whether the joint distribution over those output/runtime pairs satisfy JOT-DP (Definition 6).

We note that a highly noisy channel may introduce substantial variability into the runtime measurements, obscuring any timing differences that the adversary might otherwise detect. In practice, the variability in the runtime measurements can depend on the adversary's position in the network relative to the DP server. For example, adversaries measuring runtime over a wide-area network connection may face challenges such as network congestion, packet loss, or jitter, which can introduce significant noise and mask observable timing differences. In contrast, an adversary co-located on the same physical machine as the DP server could obtain precise runtime measurements with minimal or no noise. Throughout this paper, we use a fairly conservative threat model, where the adversary is on a separate physical machine, but resides within the same datacenter as the DP server.

4 Timing Attacks

In this section, we describe the various timing side-channels that we investigated across the surveyed libraries. We evaluated our attacks using two t2.large servers on Amazon AWS within the same region but on different physical machines and an average ping latency of 1.13ms and a standard deviation of 0.35ms. In the context of our experiments, the execution environment therefore includes network conditions such as latency and jitter, shared virtualized cloud hardware resources, and the microarchitectural characteristics inherent to the specific AWS instances. Thus, our execution environment reflects a realistic networked deployment scenario rather than an isolated or tightly controlled computational setting.

To select the libraries and specific functions targeted in our analysis, we focused on those most commonly employed in DP algorithm design and data processing workflows. In particular, we prioritized transformations and measurements frequently used in real-world data analysis pipelines, including filtering datasets, sorting operations, computing and releasing noisy aggregates such as sums and counts, and performing private partition selection. These criteria reflect realistic usage patterns in typical implementations of DP.

4.1 Attacking Sorting Transformations

We begin with a simple example that demonstrates how seemingly innocuous computations have runtime distributions that, if observed, can result in blatant violations of differential privacy. Consider the task of sorting a dataset, which is used for example as a subroutine in the DP trimmed mean estimator [8] and the smooth-sensitivity median algorithm [23]. Depending on the sorting algorithm, the runtime of this mechanism can be highly variable on adjacent datasets. For example, suppose we have two datasets $X = [1, 2, \ldots, n]$ and $X' = [n+1, n, \ldots, 2]$ that are adjacent under the change-one distance metric, i.e., you can obtain X' from X by changing the element 1 to $n+1$ and vice versa. Applying the simple bubble sort algorithm on X will result in $n-1$ comparisons since the dataset is already in sorted order. However, running bubble sort on X' results in $n \cdot (n-1)/2$ comparisons. Therefore, as n grows large, the difference between the number of comparisons (and hence the runtime) becomes more pronounced.

To demonstrate a timing side channel arising from data sorting, we implemented the DP trimmed mean algorithm [8]. This algorithm first sorts the input dataset and then discards the bottom and top α-fraction of values. It then applies the standard Laplace mechanism with appropriately scaled noise to release a DP mean. For adding Laplace noise, we relied on the diffprivlib library; however, similar functionality could be implemented using OpenDP or PyDP. Importantly, the timing vulnerability originates from the sorting subroutine itself and is therefore independent of the choice of library.

To evaluate this timing vulnerability, we constructed two adjacent datasets: $X = [1, \ldots, 20000]$ and $X' = [20001, 20000, \ldots, 2]$, and set the privacy parameter to $\varepsilon = 0.1$. A client program deployed on an AWS server issued requests to compute the DP trimmed mean on these datasets, and the server returned the corresponding outputs. Despite the modest dataset size, the timing signal was clearly detectable over the network: for X, the average response time was 5.3 milliseconds ($\sigma = 0.9$ms), while for X', the average was 8.0 milliseconds ($\sigma = 0.9$ms). Such a timing difference is easily detectable in practice, especially within data centers or over short network paths, where typical round trip latencies are often under a millisecond (e.g., AWS intra-region latencies frequently range below 1ms). In Section 5, we quantify the resulting degradation in privacy due to this timing side channel.

Although this sorting-based attack may appear contrived due to its reliance on specially structured inputs, it demonstrates how observable timing side channels arising from seemingly innocuous data processing operations can violate DP guarantees. The attack may not always appear to give an adversary much power, since exploiting it requires inputs with particular structure. However, an adversary can leverage this same side channel to perform more targeted attacks that identify whether a given individual is in the dataset. For example, consider an *equality check* transformation maps each record in the dataset to a binary value that indicates whether it equals a target record, e.g., checking if the record belongs to a user Alice. If Alice's record is present at the beginning of the dataset, the result is a binary vector with a leading 1 followed by $n-1$ zeros. If Alice is absent, the result is an all-zero vector. An adversary can then issue a query that chains this equality transformation with a measurement that performs sorting, such as the trimmed mean. The runtimes will diverge since the all-zero vector sorts quickly, while the presence of a 1 requires additional comparisons to place it in order. Thus, by chaining the equality transformation with a sorting-based measurement, an adversary can turn a weak ordering leak into a direct membership inference attack.

Moreover, this attack highlights important subtleties surrounding the choice of input metric. In particular, the runtimes of many common data processing operations, such as sorting, are inherently sensitive to the ordering of the input dataset. Thus, it may be more natural to reason about timing differences[3] with respect to an ordered adjacency metric (indeed, discrepancies between ordered and unordered metrics have previously caused underestimation of global sensitivity in DP libraries [9]). To illustrate the implications of metric choice, observe that under the conventional

[3]Recent proposals for constructing JOT-DP programs (Definition 6) rely precisely on this notion of *timing stability*, which quantifies how much a program's runtime can vary between neighboring datasets according to a chosen input metric.

unordered change one adjacency metric (Definition 1), the datasets considered earlier differ by exactly one element, so $d_{\text{CO}}(X, X') = 1$. However, under an ordered adjacency metric such as the Hamming distance, these same datasets appear substantially different, with $d_{\text{Ham}}(X, X') = 20000$, perhaps justifying the observed runtime differences. Nevertheless, minimal changes in ordering can still induce large timing variations: consider datasets $X = [1, 2, \ldots, 20000]$ and $X' = [20001, 2, \ldots, 20000]$, which differ by exactly one element in exactly one position, making them neighbors under the Hamming metric, i.e., $d_{\text{Ham}}(X, X') = 1$. Despite this small difference, the bubble sort subroutine performs an additional $O(n)$ comparisons on X' due to the placement of the maximum element. Empirically, we observed an average response time of 4.0 ms ($\sigma = 1.3$ ms) for X, increasing to 4.9 ms ($\sigma = 0.9$ ms) for X', clearly demonstrating noticeable runtime differences.

Finally, we emphasize that this sorting timing side channel also arises in the unbounded DP setting and generally affects all comparison based sorting algorithms, not only bubble sort. Executing a comparison based sorting algorithm on datasets of size n versus $n + 1$ typically results in additional comparison operations for the larger dataset. Although distinguishing the timing difference due to a small number of additional comparisons may be infeasible in practice, larger differences, such as distinguishing datasets of size n from datasets of size $n + c$ for some small constant c, could be feasible. Such timing differences could be leveraged to violate the group privacy guarantees of DP.

4.2 Attacking Partition Selection

In private data analysis, it is often necessary to select a set of partitions from the data while preserving individual privacy. Releasing aggregate statistics at the group level, for instance, can inadvertently reveal individual membership simply through the inclusion of a group. To mitigate such privacy risks, DP libraries use *private partition selection* mechanisms, designed to identify groups within a dataset that contain enough records to justify their inclusion while still adhering to DP guarantees.

Both PyDP and OpenDP support private partition selection via Laplace thresholding. At a high level, this method computes the count of records for each potential group, adds Laplace noise scaled to $1/\varepsilon$ to each count, and retains only groups whose noisy counts exceed a predefined threshold τ. Groups not meeting this noisy threshold criterion are excluded from subsequent analyses to protect individual privacy. We give a pseudocode description of Laplace thresholding in Figure 3.

Unfortunately, the Laplace thresholding mechanism introduces a subtle timing side channel. In particular, the computational overhead associated with counting, noise addition, and threshold comparisons scales with the number of distinct groups in the dataset. For example, consider two neighboring datasets: $X = [A, A]$ and $X' = [A, B]$. On X, the mechanism computes and thresholds a noisy count for group "A" only. In contrast, on X', it must additionally compute and threshold a noisy count for group "B". This results in additional computation and, consequently, a longer runtime. These runtime differences, driven purely by the number of distinct groups, can leak sensitive information about the dataset's composition. For instance, if the mechanism returns only a single group but exhibits an unusually long execution time, it may reveal that multiple other groups were present but failed the threshold check.

```
hist = {}
for record in data:
    cat = get_category(record)
    if hist[cat] = null:
        hist[cat] = 0

    hist[cat] = hist[cat] + 1

for cat in list(hist.keys()):
    noisy = hist[cat] + laplace(scale)
    if noisy < threshold:
        del hist[cat]
    else:
        hist[cat] = noisy

return hist
```

Figure 3: Laplace thresholding pseudocode. The mechanism first builds a histogram of category counts, then adds Laplace noise to each count and removes categories with noisy counts below the threshold.

We experimentally validate the existence of this timing side channel in both PyDP and OpenDP. For PyDP, we utilize the PipelineDP framework, relying on PyDP's partition selection implementation. Adapting PipelineDP's official introductory example [26], we conducted experiments varying the number of distinct groups in datasets containing a fixed total of 100 records. A similar experimental protocol was applied to OpenDP's Laplace thresholding method. Our results, depicted in Figure 4, demonstrate a clear linear relationship between runtime and the number of groups present within the dataset. The x-axis represents the number of partitions filtered out by the mechanism (specifically, groups with only a single record).

To evaluate a timing attack on the partition selection mechanism, we configured the OpenDP Laplace thresholding mechanism with privacy parameters $\varepsilon = 0.01$ and $\delta = 10^{-40}$. We considered adjacent datasets of 5000 records where in X all records belonged to a single category and yielded one partition, while in X' an additional record introduced a second partition with count 1 that was, with high probability, removed by the thresholding step. Empirically, we observed that the average response time for X was 3.9 ms ($\sigma = 0.89$ ms), while for X' it increased to 4.3 ms ($\sigma = 1.14$ ms) in our AWS environment. In Section 5, we show that even such small, yet detectable timing differences can lead to additional privacy loss.

Much like the attack described in §4.1, the Laplace thresholding mechanisms fail to protect information about the *size* of the dataset when their runtime distribution is observable. This leakage can be exploited by a malicious analyst to infer the approximate number of low-frequency groups in the data. Furthermore, even in the bounded DP setting, enforcing constant-time execution by padding to the worst-case runtime for inputs of length n may be infeasible if the set of groups is unbounded (e.g., when each record belongs to an infinite domain such as the set of all strings). We provide additional discussion of mitigation strategies in Section 6.

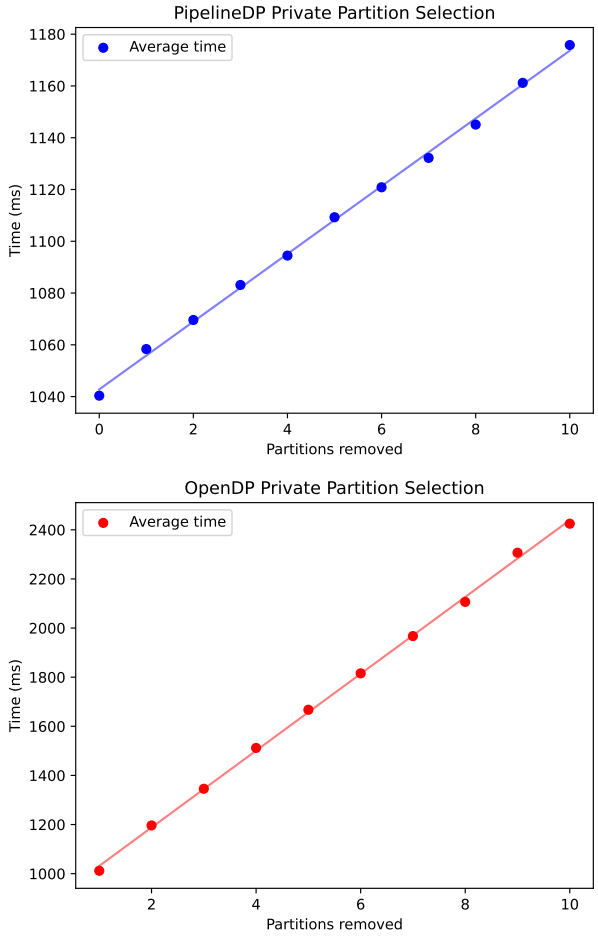

Figure 4: Timing side channels in the private partition selection mechanisms in PipelineDP (PyDP) and OpenDP. The runtime of the mechanisms are linearly correlated with the total number of partitions (groups) in the dataset containing 100 records.

4.3 Attacking Floating-Point Multiplication

Prior work by Andrysco et al. demonstrated that mainstream CPU architectures exhibit timing discrepancies between subnormal and normal floating-point operations [3], and such discrepancies could be exploited to break DP systems that were specifically designed to prevent timing attacks [14]. For example, on modern CPUs, adding a subnormal float to a normal float takes more CPU cycles than adding two normal floats. While the raw wall-clock difference between these operations may only amount to tens of nanoseconds, the difference can be amplified when the operations are performed repetitively in loops. Andrysco et al.'s attack on the Fuzz DP system worked as follows: the attacker issues a query that (1) transforms the target record (if present) to a subnormal floating-point value (e.g., "1e-308") and (2) transforms all other records to a normal floating-point value of 0.0. The second part of the query computes a private sum over the transformed rows. During this summation,

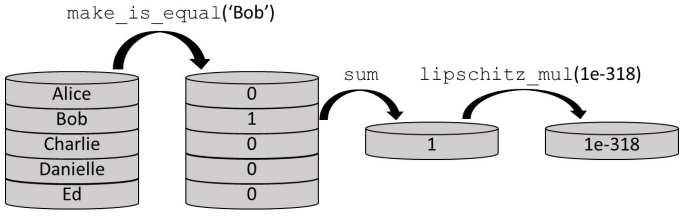

Figure 5: The subnormal probe gadget. We chain together an `is_equal` transformation with the `sum` and `lipschitz_mul` transformations to create a single aggregate value. The aggregate value equals the subnormal float 10^{-318} if the dataset contains the target record and 0.0 otherwise.

Fuzz would either repeatedly add subnormal floats or normal floats, creating a measurable timing signal detectable by the attacker.

Given that the OpenDP programming framework supports subnormal floating-point data types, we investigated whether OpenDP exhibits similar floating-point induced timing side channels. In our evaluation, we successfully replicated an attack similar to that of Andrysco et al., enabling us to infer the presence of a target record in a dataset. Notably, our approach does not depend on user-defined functions; instead, it leverages only core transformations available within the OpenDP library. Central to our attack is the use of a c-stable *Lipschitz multiplication* transformation, characterized by a Lipschitz constant c. This transformation multiplies the output of a preceding transformation (such as a sum or mean) by the constant factor c. Its stability property ensures that bounded input variations yield controlled and bounded output variations, precisely quantified by the Lipschitz constant.

The attack works as follows. First, assume the attacker can construct a query using any of the available transformations within the OpenDP library (which include the *is_equal*, *sum*, and *lipschitz_mul* transformations). Then the attacker defines a *probe gadget* made up of the following chain of transformations (we use pseudocode for brevity)[4]:

```
probe = (
    is_equal(target) >>
    sum >>
    lipschitz_mul(1e-318)
)
```

The `is_equals` transformation converts the dataset into a vector of 1 values (indicating that the dataset row equals `target`) and 0 values (indicating that the dataset row $\neq$ `target`). The `sum` transformation then sums the vector entries. Assuming the dataset consists of unique rows, the `sum` transformation produces the aggregate value 1 if the target record is present and 0 otherwise. Finally, the resulting sum is multiplied by the subnormal float value 1e-318. Thus, the probe gadget transforms the dataset into the subnormal float 1e-318 if the target record is present, and the normal float value 0.0 otherwise.

[4]We use the » operator to indicate chaining transformations.

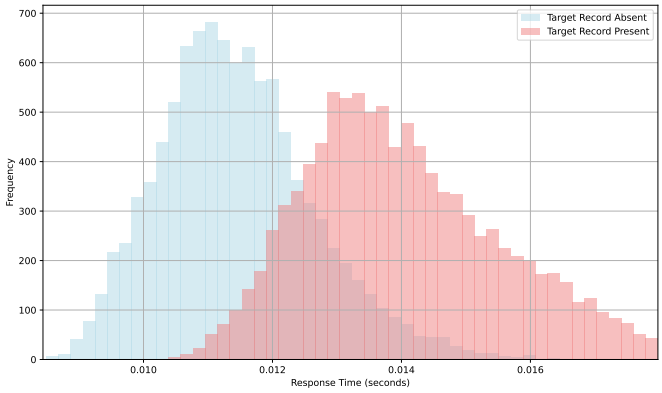

Figure 6: Timing attack on OpenDP leveraging a timing side channel in floating-point arithmetic on modern CPUs. The experiment sets up a subnormal floating-point value if the target record is present in the dataset, and a normal floating-point value otherwise. The attack then measures the runtime of repeatedly multiplying the resulting value by a constant.

The attacker then chains the probe gadget with an arbitrarily long chain of Lipschitz multiplication transformations, which we refer to as a *test gadget*:

```
test = (
    lipschitz_mul(1.0) >>
    lipschitz_mul(1.0) >>
    ...
    lipschitz_mul(1.0)
)
```

As described earlier, the `lipschitz_mul` transformation takes a result and multiplies it by a constant (in this case, the constant 1.0). Thus, the above test gadget performs repeated multiplications by the floating-point value 1.0. Finally, the attacker can chain together the *probe* and *test* gadget with a `laplace` measurement[5] to produce a differentially private output.

```
mechanism = probe >> test >> laplace
```

Since repeated multiplications of $1.0 \times$ 1e-318 take longer than those of 1.0×0.0, we can deduce the presence of the target record by analyzing the runtime of the entire multiplication chain.

Our evaluation, illustrated in Figure 6, highlights the runtime discrepancies observed when the mechanism operates on a dataset X (where the target record is absent) versus a neighboring dataset X' (where the target record is present). The experimental setup involved a chain consisting of 25,000 Lipschitz multiplications. When executed on X, the mechanism exhibited an average response time of 11.4 ms ($\sigma = 1.20$ ms), while for X' the average increased to 14.1 ms ($\sigma = 1.57$ ms). This 2.6 ms timing gap constitutes a measurable side channel, allowing an adversary to distinguish between the two datasets with high confidence. Notably, the severity of the timing side channel increases with the length of the Lipschitz multiplication chain, amplifying the observable timing gap between adjacent inputs.

[5]The choice of the Laplace mechanism is arbitrary and the side channel may be applied more generally to other mechanisms.

At first glance, this attack may seem contrived, as it requires the adversary to construct a long chain of repeated multiplications. However, the attack relies solely on standard transformations available within the OpenDP core, and such transformation chains can plausibly arise in practice. For example, this situation could occur when a data curator allows analysts to issue arbitrary DP queries, even if those queries are restricted to using core transformations provided by the OpenDP library. While OpenDP could potentially detect and restrict the construction of such chains, it remains unclear how to comprehensively identify every transformation combination that might introduce timing side channels.

Finally, while our attack builds on the same technique introduced by Andrysco et al., our attack is arguably more practical and severe. The timing signal in the Fuzz attack was highly dependent on dataset size and the position of the target record. In contrast, our attack remains effective on datasets as small as one record and is entirely independent of the target's position within the dataset. As a result, the attack can be scaled to perform full dataset enumeration by iterating over candidate target records and testing for their presence individually.

4.4 Attacking Filtering Transformations

We now detail an attack on mechanisms that process specific subsets of their input. As an illustrative example, consider a data analyst interfacing with a healthcare dataset that wants to compute the number of individuals aged 30 years and younger that have a particular disease. To achieve this, the analyst might run a DP mechanism that:

- filters the dataset to isolate rows corresponding to individuals under 30 years old, and
- computes a DP sum over the disease column of this filtered subset.

Unfortunately, the computation in step (2) can have a runtime that is highly correlated with the filtered dataset size produced by step (1). This correlation could inadvertently reveal the number of individuals in the dataset who are under 30 years old.

We demonstrate this timing side channel using OpenDP,[6] which has recently added support for Polars data frames. Specifically, we adapted example code, taken from the OpenDP documentation [24], which applies a filtering transformation to an example microdata dataset from the EU Labor Force Survey (we provide a pseudocode description in Figure 7). The mechanism first filters individuals whose usual weekly working hours (`HWUSUAL`) are below 30 and then computes a DP sum over the `HWUSUAL` column for the remaining records. We then executed the mechanism on adjacent datasets X and X', where X contains 5,000 individuals who usually work fewer than 30 hours per week, and X' contains 0 such individuals. We set the filtering criteria to select only rows where `HWUSUAL` < 30 and configured the chained DP sum with privacy parameter $\varepsilon = 10^{-5}$. Consequently, the filtered dataset derived from X contained exactly 5,000 records, while the filtered subset of X' was empty. Therefore, the DP summation loop executed over 5,000 rows for dataset X and over zero rows for dataset X', resulting in a measurable timing difference. Importantly, under a group privacy guarantee, one would

[6]Neither diffprivlib nor PyDP include built-in functionality for filtering; however, both libraries support filtering datasets using native Python.

```
filtered_df = (
    dataset.lazy()
    .filter(pl.col("HWUSUAL") < 30)
    .collect()
)

context = dp.Context.compositor(
        data=filtered_df,
        ...
)

query_total_hours_worked = (
    context.query()
    .with_columns(pl.col.HWUSUAL.cast(int).fill_null(0))
    .select(pl.col.HWUSUAL.dp.sum((0, 80)))
)
query_total_hours_worked.release().collect()
```

Figure 7: Pseudocode for filtering a dataset and releasing a differentially private sum on the resulting subset.

expect these datasets to be protected under ε-DP with $\varepsilon = 10^{-5} \times 5{,}000 = 0.05$. However, as we show in (§5.1), the observed timing difference allows for distinguishing the presence or absence of entire groups of individuals.

We note that in practice the timing signal from a single DP sum may not be strong enough to exploit directly. In our experiments, we were only able to use it to violate group privacy. However, the signal can be amplified, for example by chaining additional row-by-row transformations whose cost scales with dataset size. Such amplification could strengthen the leakage enough to threaten individual privacy, particularly if filtering criteria are chosen to isolate small groups or even single records. The deeper issue is that the DP sum transformation is assumed to provide guarantees in the *unbounded* setting, where dataset size itself must be protected, but these guarantees break down once runtimes are observable—effectively reducing the mechanism to a bounded-DP one that leaks input size. This risk is further exacerbated in adaptive query systems that cache intermediate results: an analyst could filter a subset once, return and cache it, and then issue subsequent queries whose runtimes scale with the subset size. By iterating on filtering criteria, an adversary could exploit these timing dependencies to infer subset sizes and potentially enumerate individual records.

4.5 Attacking Randomized Response

Boolean randomized response is one of the more basic and well-known algorithms in the DP toolkit. The technique is commonly used in the setting of local DP where individual users each run a query on their own data and randomize the output *before* sending the private outputs to a central server for additional analysis. For example, consider the setting in which a server polls many remote users asking for a single bit of information, e.g., whether the individual is currently feeling sick (such a system may be used for real-time distributed analytics to determine things like the spread of disease throughout a community [29]). Each user runs a local randomizer on their private bit before sending the response to the server. Unfortunately, if the runtime of the randomizer depends on either the private data or the internal randomness used privatize the output (or both), then the server may be able to learn an individual's true data.

```
def rand_resp(x, eps, delta):
    u = random() * (exp(eps) + 1)
    if u > exp(eps) + delta:
        x = 1 - x

    return x
```

Figure 8: Pseudocode for the randomized response mechanism implemented in `diffprivlib`.

We investigated the built-in implementations of randomized response in both diffprivlib and OpenDP. The diffprivlib library performs randomized response using the process[7] in Figure 8. At a high level, the algorithm first samples a uniform random variable $X \in [0, 1]$. If $X \cdot (e^{\varepsilon} + 1) > e^{\varepsilon} + \delta$, then the program branches and flips the true bit. Otherwise, the true bit is returned. Note that the branching behavior introduces additional code that only executes depending on the outcome of the random variable X. Therefore, if you could learn the *exact* runtime of the program, along with the DP output y, you could determine whether $x = y$ or $x = 1 - y$.

We experimentally verified this timing side channel in diffprivlib's randomized response mechanism (Figure 9). The running time of the mechanism differed by approximately 60 nanoseconds on average due to the additional branching logic when the input bit is flipped. While such a small timing difference could in theory violate differential privacy, accurately measuring a nanosecond-level signal from a single sample is highly improbable. An attacker would need to collect a large number of samples for this attack to become practical, and in over-the-network scenarios natural sources of noise would almost certainly mask the effect. Nonetheless, the side channel could be eliminated entirely by adopting constant-time programming practices.

By contrast, the OpenDP implementation of randomized response already employs constant-time programming. Specifically, it samples a Bernoulli random variable with probability p in constant time and then XORs the input bit with the result, ensuring that the runtime is independent of the internal randomness used to privatize the output. In our experiments, the OpenDP mechanism exhibited constant-time behavior as intended by its developers (Figure 9).

5 Auditing Timing Attacks

In this section, we introduce an auditing technique for detecting violations of DP when a program's runtime is observable to an adversary. Our method estimates lower bounds on a program's JOT-DP privacy parameters by analyzing the joint distribution of its outputs and runtimes. Concretely, we extend the statistical auditing framework of Jagielski, Ullman, and Oprea [16] to the joint random variable formed by both the program's output and its execution

[7] We've modified the code for brevity.

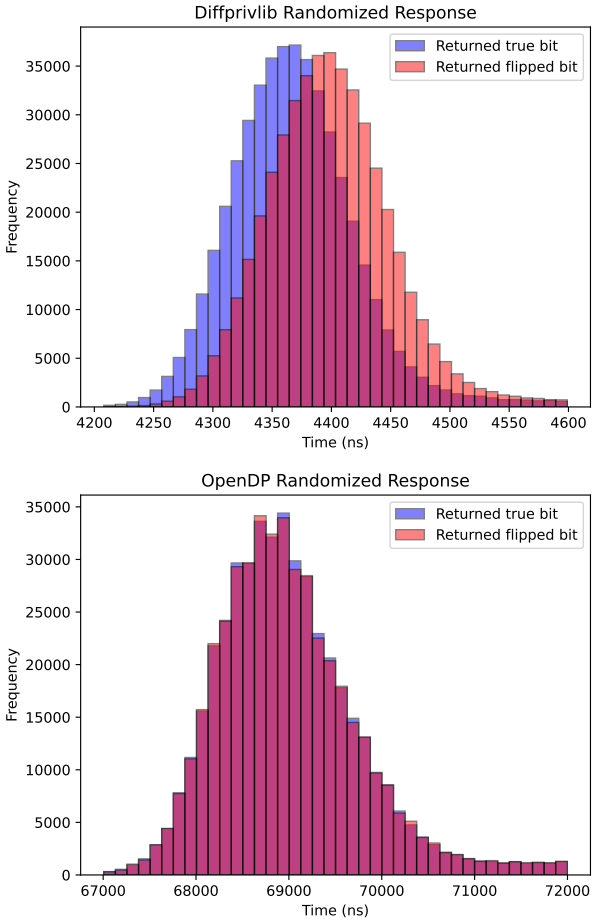

Figure 9: Running times of the randomized response mechanisms in the diffprivlib and OpenDP libraries. The mean execution time of the `randomise` function in diffprivlib differs by approximately 60 nanoseconds. The OpenDP randomized response mechanism exhibits constant runtime behavior.

time. While the extension is conceptually straightforward, auditing in this setting introduces a new set of challenges. First, unlike program outputs,[8] runtimes can vary significantly across execution environments, and empirical probabilities must often be estimated from drifting execution environments whose precise states may be unknown or difficult to accurately measure. Second, execution time lies in a (nearly) continuous domain, complicating direct estimation of event probabilities. Fortunately, this second challenge is easily addressed by considering events defined by a threshold on runtime (e.g., the frequency of executions that exceed a particular timing threshold). This converts the auditing problem from estimating probabilities over a continuous domain into a simpler binary classification task.

Our auditing technique is adapted from that of Jagielski et al. as follows. First, the program is executed multiple times to obtain the sets of output/runtime pairs $S = \{(y_i, t_i)\}_{i=1}^m$ and $S' = \{(y'_i, t'_i)\}_{i=1}^m$ on adjacent datasets X and X' respectively. Using the output and runtime pairs, the following empirical estimates are computed:

$$\hat{p} = \frac{1}{m} \cdot \sum_{i=1}^{m} \mathbb{I}\{(y_i, t_i) \in O\}$$

and similarly for X'

$$\hat{p}' = \frac{1}{m} \cdot \sum_{i=1}^{m} \mathbb{I}\{(y'_i, t'_i) \in O\}$$

for some set $O \subseteq \mathcal{Y} \times \mathcal{T}$. Confidence intervals $\hat{p}_u$ and $\hat{p}_\ell$ are computed such that the true probability $\Pr[(P(X, \mathsf{env}), T_P(X, \mathsf{env})) \in O]$ is within the intervals $[\hat{p}_\ell, \hat{p}_u]$ with confidence $1 - \alpha/2$ for each bound (and similarly for $\hat{p}'_u$ and $\hat{p}'_\ell$). Using these statistical estimates, we can establish that with probability at least $1 - \alpha$, the program P does *not* satisfy (ε, δ)-JOT-DP (Definition 6) for any $\varepsilon < \ln((\hat{p}_\ell - \delta)/\hat{p}'_u)$.

When auditing runtime measurements, the choice of the set O directly impacts our ability to reliably detect violations of differential privacy. Since runtime values are drawn from a continuous (or nearly continuous) domain, attempting to estimate probabilities for overly precise or fine grained events, such as exact runtime values, can lead to misleading differences, as certain runtime values may appear in one dataset but not in the other. Moreover, estimating such precise probabilities is statistically unstable because empirical frequencies of exact continuous values typically remain at or near zero. To avoid this issue, we select a threshold T and define the event O as the set of runtime measurements either less than or greater than T. This effectively transforms the auditing task over a continuous domain into a simpler binary classification problem, where the goal is to measure how often execution times fall above or below a chosen threshold on adjacent datasets. The threshold is selected using a held out portion of the data to avoid bias from data-dependent threshold selection.

In addition to the challenge of defining meaningful events, auditing runtime also requires addressing the dependence of empirical runtime probabilities on the execution environment env. Since runtimes can differ significantly across environments and since the precise state of these environments might drift or be difficult to accurately measure, maintaining a perfectly fixed execution environment across multiple runs is typically impractical. Thus, our empirical probability estimates are effectively computed over a mixture of execution environments. Fortunately, Lemma 1 ensures that fixing the execution environment is unnecessary, as the joint output and timing privacy guarantees remain valid even when evaluated over mixtures of potentially unknown or adversarially selected environments.

Lemma 1. *Let $P : \mathcal{X} \times \mathcal{E} \to \mathcal{Y} \times \mathcal{E}$ be an (ε, δ)-JOT-DP program. Then for all adjacent datasets $X, X' \in \mathcal{X}$, all sets $E = \{(\mathsf{env}_j, \mathsf{env}'_j)\}_{j \in J}$ of pairs of input-compatible execution environments, all probability distributions $\mathcal{J}$ over the index set J, and all measurable sets $O \subseteq \mathcal{Y} \times \mathcal{T}$, we have*

$$\Pr\Big[(Y, T) \in O\Big] \leq e^{\varepsilon} \cdot \Pr\Big[(Y', T') \in O\Big] + \delta$$

where $j \sim \mathcal{J}$, $Y = P(X, \mathsf{env}_j)$, $T = T_P(X, \mathsf{env}_j)$, $Y' = P(X', \mathsf{env}'_j)$, and $T' = T_P(X', \mathsf{env}'_j)$.

[8] Indeed, program outputs must be environment-independent for standard composition theorems to hold.

Proof. Since P is (ε, δ)-JOT-DP, by Definition 6 for every pair $(\mathsf{env}_j, \mathsf{env}'_j)$ of input-compatible execution environments and for all adjacent $X, X' \in \mathcal{X}$, we have

$$\Pr\left[(Y_j, T_j) \in O\right] \leq e^{\varepsilon} \Pr\left[(Y'_j, T'_j) \in O\right] + \delta$$

where $Y_j = P(X, \mathsf{env}_j), T_j = T_P(X, \mathsf{env}_j)$, and Y'_j and T'_j are defined similarly.

Let J be a random index with distribution $\mathcal{J}$. Then, by taking the expectation over j, we obtain

$$\begin{aligned}\Pr\left[(Y, T) \in O\right] &= \sum_j \Pr[J = j] \cdot \Pr\left[(Y_j, T_j) \in O\right] \\ &\leq \sum_j \Pr[J = j] \cdot \left(e^{\varepsilon} \cdot \Pr\left[(Y'_j, T'_j) \in O\right] + \delta\right) \\ &= e^{\varepsilon} \cdot \Pr\left[(Y', T') \in O\right] + \delta\end{aligned}$$

□

The above result indicates that if a program is (ε, δ)-JOT-DP, its privacy guarantees hold regardless of how the execution environment is chosen during auditing.

Improving the Empirical Estimate. We can take advantage of the known output PMF for a DP program to improve the estimates $\hat{p}$ and $\hat{p}'$. Instead of estimating $\hat{p}$ and $\hat{p}'$ purely from empirical data, we use the exact PMFs[9] on the outputs of the program to calculate:

$$\hat{p} = \Pr[P(X, \mathsf{env}) \in \pi_1(O)] \cdot \hat{p}_t$$
$$\hat{p}' = \Pr[P(X', \mathsf{env}) \in \pi_1(O)] \cdot \hat{p}'_t$$

where $\pi_1(O) = \{y \in \mathcal{Y} : \exists t \in \mathcal{T}, (y, t) \in O\}$ and $\hat{p}_t$ and $\hat{p}'_t$ are empirical estimates of the conditional runtime distributions. Specifically, $\hat{p}_t$ estimates:

$$\Pr[T_P(X, \mathsf{env}) \in \pi_2(O) | P(X, \mathsf{env}) \in \pi_1(O)]$$

and $\hat{p}'_t$ estimates:

$$\Pr[T_P(X', \mathsf{env}) \in \pi_2(O) | P(X', \mathsf{env}) \in \pi_1(O)]$$

where $\pi_2(O) = \{t \in \mathcal{T} : \exists y \in \mathcal{Y}, (y, t) \in O\}$. This allows us to incorporate known properties of the program's output distribution while empirically estimating the program's conditional runtime distribution.

The auditing framework described above is used throughout this paper to assess whether an implementation fails to satisfy JOT-DP for a given set of privacy parameters. We present a couple of observations. First, when the output set $\pi_1(O)$ covers the entire output domain of the program, the auditing framework measures the privacy guarantees provided by the program when *only* its runtime is observable. In many cases, this alone is sufficient to show that a program does not meet the desired level of privacy, as runtime distributions can vary significantly on neighboring inputs.

Second, we note that the program's execution environment can influence the observed runtimes, as captured in Definition 6. Ideally, the environment for auditing timing attacks will closely mirror the resources and constraints of the production environment. Specifically, Definition 6 quantifies privacy across all pairs of execution environments within a set $\mathcal{E}$. If $\mathcal{E}$ is highly restricted (e.g., contains only a single execution environment[10]) it becomes easier to obtain a tight lower bound on the program's privacy parameters. However, it may be challenging to predict the exact execution environment, particularly in interactive settings where the system may handle multiple queries and update its internal state over time. For instance, if a query places the system in a rare but valid architectural state that induces a severe timing side channel, the program might fail to achieve JOT-DP for any reasonable setting of ε and δ. Detecting this during auditing may be unlikely if the specific architectural state that triggers the timing side channel cannot be reproduced. In practice, this implies that the empirical lower bound is only useful for *verifying* JOT-DP properties if one can ensure that the execution environments used in production are equivalent to those in the audit (otherwise, the real lower bound might be significantly worse than the bound reported by the audit). Therefore, carefully controlling and replicating the execution environment can be crucial in detecting timing vulnerabilities during auditing. On the other hand, we leverage the auditing framework to *verify timing attacks*; that is, to demonstrate that a given program fails to achieve timing privacy with high probability for a specified setting of ε and δ.

5.1 Evaluation of Attacks

We evaluate our timing attacks using the auditing algorithm described in the previous section. Across all experiments, we observe a measurable degradation in privacy, with the auditor returning lower bounds ε_{LB} that often far exceed the nominal privacy parameters configured for the mechanism (see Table 1). Except for the randomized response experiments (§4.5), all timing measurements were collected over a network connection, demonstrating that these attacks are practical in real-world settings. We emphasize that the mechanisms were configured in the high privacy (low ε) regime where one would expect little or no information leakage from outputs alone, yet our audits reveal that runtime effects induce significant additional leakage. We discuss the practical implications of this gap in the next section.

In concrete terms, the strongest leakage appeared in our audits of the DP Trimmed Mean and Lipschitz Counting mechanisms, where we measured effective privacy losses of $\varepsilon_{\text{LB}} = 6.6$ and $\varepsilon_{\text{LB}} = 5.9$, despite nominal configurations of $\varepsilon = 0.1$ for both mechanisms, respectively. In both cases, the structure of the algorithm allows the side channel to become arbitrarily severe. For the trimmed mean attack, the timing gap scales with dataset size and row ordering, while in the Lipschitz counting attack the adversary can chain multiple gadgets to magnify the effect. Even in the partition selection, filtered sum, and randomized response settings where the reported ε_{LB} values may appear more modest in absolute terms, the effective privacy degradation is dramatic, often exceeding the configured privacy parameters by an order of magnitude. As we will discuss in the next section, such leakage could be exploited by a determined network adversary, even if the adversary is required to collect and average multiple timing measurements.

[9] However, one might still prefer to empirically estimate the program's output distribution to detect additional vulnerabilities, such as floating-point issues [22].

[10] This might be achievable, for instance, by consistently executing the program from a clean micro-architectural state in an isolated environment with no competing processes

Mechanism	Privacy Parameters	Audit Lower Bounds	Over-the-Network
DP Trimmed Mean (§4.1)	$\varepsilon = 0.1$	$\varepsilon_{\text{LB}} = 6.6$	Yes
Partition Selection (§4.2)	$\varepsilon = 0.01, \delta = 10^{-40}$	$\varepsilon_{\text{LB}} = 0.3$	Yes
Lipschitz Mult (§4.3)	$\varepsilon = 0.1$	$\varepsilon_{\text{LB}} = 5.9$	Yes
Filtered Sum (§4.4)	$\varepsilon = 0.05$	$\varepsilon_{\text{LB}} = 1.58$	Yes
Randomized Response (§4.5)	$\varepsilon = 0.01$	$\varepsilon_{\text{LB}} = 1.17$	No

Table 1: Summary of Timing Side-Channel Attacks. The ε_{LB} parameter indicates the lower bound on the mechanism's privacy parameters returned by our auditing algorithm, and were estimated with 99% confidence intervals. The "over-the-network" column indicates whether the auditing lower bound was established by collecting measurements over the network, or directly on the target machine. For the filtered sum mechanism, we audited against a group privacy bound of $\varepsilon = 0.05$. For randomized response, we evaluated diffprivlib's implementation.

6 Discussion

We have shown that timing side channels are pervasive across implementations of differential privacy. Mechanisms frequently and trivially leak sensitive information about their inputs through observable runtime behavior, most commonly revealing the input length. Although this leakage might initially seem harmless, our findings demonstrate that chaining multiple transformations can amplify seemingly minor leaks into severe privacy vulnerabilities. Moreover, we observe that runtime distributions of DP algorithms are often significantly influenced by the ordering of input data. Recent work has already highlighted instances where DP libraries underestimated global sensitivity due to ambiguities between ordered and unordered dataset metrics. When considering defenses against timing attacks specifically, our analysis underscores the necessity of adopting ordered metrics, as runtime variations can heavily depend on data ordering even when the output remains indistinguishable.

In practice, one challenge for an adversary is that exploiting runtime leakage over a network often requires many repeated measurements to overcome natural timing noise. However, this requirement does not preclude attacks. An adversary could, for instance, allocate only a negligible fraction of the privacy budget to each query. In that case, the outputs themselves would be effectively uninformative, but the adversary could still harvest runtime information. By issuing a large number of such low-budget queries, the adversary can collect the timing samples needed to reliably exploit the side channel.

We now discuss potential mitigations against timing attacks.

Constant-Time Programming. A standard defense against timing attacks is to enforce constant-time computation by padding execution to a worst-case runtime across all inputs. However, achieving truly constant-time behavior is challenging due to inherent variability at the microarchitectural level of modern CPUs. Furthermore, constant-time implementation typically requires writing specialized code [10] or using processors that forego performance optimizations such as speculative execution [33]. Consequently, constant-time defenses tend to result in slower, less flexible systems, which might limit their adoption in performance-critical DP applications.

In the unbounded setting of DP, constant-time programming is perhaps an even more *unattractive* solution. In this setting, the developer would need to choose an upper bound on dataset size and pad execution to the worst-case runtime on inputs of that size, regardless of the true size of the input dataset. This not only imposes severe performance overhead but also undermines the scalability of the system, since the chosen bound must accommodate even rare, extremely large inputs.

Randomizing the Runtime. As discussed in (§7), recent theoretical frameworks address timing attacks by explicitly incorporating runtime into the DP guarantee. Under this approach, a program satisfies JOT-DP (Definition 6) if the joint distribution of its output and runtime achieves differential privacy. If the program's *timing stability* is bounded, differential privacy can be enforced by adding a carefully calibrated delay before releasing outputs [28]. Applying this theoretical framework to practical DP implementations running on real-world hardware is a promising but challenging future direction. Accurately quantifying timing stability in practice will likely require a combination of static and dynamic program analysis, and since timing stability inherently depends on hardware specifics, bounds would need to be determined separately for each deployment environment.

Nevertheless, this runtime-randomization approach seems especially suitable for mitigating several attacks identified in our analysis. In particular, many of our demonstrated attacks result in runtime differences on the order of milliseconds, suggesting that relatively modest delays may suffice to achieve JOT-DP. Thus, randomizing runtime could mitigate timing vulnerabilities without incurring the large performance overhead typically associated with strict constant-time programming.

Leveraging Natural Timing Variation. Program runtimes inherently exhibit variability from concurrent system processes, network jitter, and hardware non-determinism. This natural timing variation could help obscure sensitive timing signals. For example, the Boolean randomized response mechanism described in (§4.5) introduces timing signals on the order of nanoseconds that are difficult to exploit without highly precise local measurements. In networked scenarios, jitter typically overwhelms these signals, significantly diminishing attack feasibility. This naturally occurring delay closely resembles timing-private delay functions [28], and

formalizing how natural timing variations could enhance privacy guarantees presents an intriguing direction for future research.

Preventing Timing Signal Amplification. Several timing attacks presented in this work rely on amplification techniques such as chaining multiple logical operations to enhance timing signals and facilitate more accurate measurements. Without amplification, attackers typically need repeated mechanism executions to achieve precise timing estimates. However, repeatedly querying a DP mechanism accumulates privacy loss through composition and may outweigh the informational gain of the timing attack.

Another attacker strategy involves issuing numerous queries, each with negligible privacy budgets. For example, requesting many responses from a Boolean randomized response mechanism with an exceedingly small privacy budget (e.g., $\varepsilon = 10^{-9}$) would yield near-random outputs, yet precise aggregate timing measurements could still reveal sensitive information. To counter such attacks, enforcing minimum privacy-budget expenditures per query ensures each query provides meaningful statistical output rather than merely probing runtime. Notably, the privacy accounting modules in the libraries we examined already implement such minimum budget constraints.

7 Related Work

Timing side channel attacks have been extensively studied over the past few decades within the applied cryptography and broader security communities. Observable runtime variations have repeatedly undermined implementations of cryptographic primitives and protocols previously thought to be secure. For example, seminal work by Kocher [19] demonstrated that timing information could be leveraged to break implementations of cryptographic algorithms such as RSA, Diffie Hellman, and DSS. Subsequent research identified practical cache timing attacks on AES [6, 27], timing attacks against TLS [1, 2], cache side channels including Flush+Reload [32], and more recent microarchitectural timing attacks such as Spectre [18] and Meltdown [20].

Given the impact timing side channels have had on cryptography, it is not surprising that they have also emerged as a threat to differential privacy. Haeberlen, Pierce, and Narayan [14] demonstrated that timing side channels could undermine the privacy guarantees of DP systems, highlighting the need for careful system design to control timing variations. They introduced the Fuzz DP system specifically to protect against these side channel attacks. The Fuzz system enables data analysts to issue arbitrary queries constructed from constant time *microqueries*, each of which operates on an individual row of the dataset, executes for a predefined constant amount of time, and returns a default value if this time bound is exceeded. Importantly, the Fuzz system is designed specifically for the bounded DP setting, in which the size of the dataset is considered public knowledge. This contrasts with some of the mechanisms discussed in this paper, which are intended to provide DP protections for the dataset size itself. Andrysco et al.[3] later identified timing vulnerabilities even in the Fuzz system, demonstrating attacks that exploited the data-dependent timing behavior of floating point operations. More recently, Jin et al. [17] analyzed timing side channels affecting the Discrete Laplace and Discrete Gaussian mechanisms. They demonstrated that the runtime of the sampling algorithms underlying these mechanisms can strongly correlate with the magnitude of the sampled noise, potentially leaking private information.

Recent work has also begun to investigate how we can protect differentially private programs against timing attacks by making the joint distribution of the output and runtime differentially private [5, 28]. Ben Dov et al. characterized which distributions can be sampled in a timing-safe manner, where the runtime of the sampler does not leak any information about the output. Their work also explored the extent to which pure DP mechanisms can be made resistant to timing attacks. Ratliff and Vadhan introduced a new definitional framework for reasoning about timing privacy, which includes the notion of *timing stability*, the timing analogue of global sensitivity. Their framework achieves timing privacy by injecting carefully calibrated delay into a program's execution prior to releasing its output, ensuring that the joint distribution of the output and runtime jointly satisfies (ε, δ)-differential privacy. They provided several proof of concept constructions of timing private DP programs in the RAM and Word RAM models of computation.

Acknowledgments

Zachary Ratliff's work is supported in part by Cooperative Agreement CB20ADR0160001 with the Census Bureau, and in part by Salil Vadhan's Simons Investigator Award. Nicolas Berrios' work was supported by OpenDP. We thank the OpenDP team, and specifically, Michael Shoemate for thoughtful discussions.

References

[1] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *2013 IEEE symposium on security and privacy*, pages 526–540. IEEE, 2013.

[2] Martin R Albrecht and Kenneth G Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of tls. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*, pages 622–643. Springer, 2016.

[3] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639. IEEE, 2015.

[4] Victor Balcer and Salil Vadhan. Differential privacy on finite computers. *Journal of Privacy and Confidentiality*, 9:2, 2019.

[5] Yoav Ben Dov, Liron David, Moni Naor, and Elad Tzalik. Resistance to timing attacks for sampling and privacy preserving schemes. In *4th Symposium on Foundations of Responsible Computing (FORC 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[6] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[7] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*, pages 355–371. Springer, 2011.

[8] Mark Bun and Thomas Steinke. Average-case averages: Private algorithms for smooth sensitivity and mean estimation. *Advances in Neural Information Processing Systems*, 32, 2019.

[9] Sílvia Casacuberta, Michael Shoemate, Salil Vadhan, and Connor Wagaman. Widespread underestimation of sensitivity in differentially private libraries and how to fix it. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 471–484, 2022.

[10] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 69–76. IEEE, 2017.

[11] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, pages 265–284. Springer, 2006.

[12] Marco Gaboardi, Michael Hay, and Salil Vadhan. A programming framework for opendp. *Manuscript, May*, 2020.

[13] Marco Gaboardi, Michael Hay, and Salil Vadhan. Programming frameworks for differential privacy. *arXiv preprint arXiv:2403.11088*, 2024.
[14] Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. Differential privacy under fire. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
[15] Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. Diffprivlib: the IBM differential privacy library. *ArXiv e-prints*, 1907.02444 [cs.CR], July 2019.
[16] Matthew Jagielski, Jonathan Ullman, and Alina Oprea. Auditing differentially private machine learning: How private is private sgd? *Advances in Neural Information Processing Systems*, 33:22205–22216, 2020.
[17] Jiankai Jin, Eleanor McMurtry, Benjamin IP Rubinstein, and Olga Ohrimenko. Are we there yet? timing and floating-point attacks on differential privacy systems. *arXiv preprint arXiv:2112.05307*, 2021.
[18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.
[19] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.
[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.
[21] Frank D McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30, 2009.
[22] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 650–661, 2012.
[23] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 75–84, 2007.
[24] OpenDP Project. Preparing microdata — opendp documentation. https://docs.opendp.org/en/stable/getting-started/tabular-data/preparing-microdata.html#Filter, 2025. Accessed: 2025-08-04.
[25] OpenMined. https://github.com/OpenMined/PyDP, 2023.
[26] OpenMined. Pipelinedp codelab examples. https://github.com/OpenMined/PipelineDP/blob/main/examples, 2023.
[27] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20. Springer, 2006.
[28] Zachary Ratliff and Salil Vadhan. A framework for differential privacy against timing attacks. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024. Forthcoming.
[29] Edo Roth, Karan Newatia, Yiping Ma, Ke Zhong, Sebastian Angel, and Andreas Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 327–343, 2021.
[30] Michael Shoemate, Andrew Vyrros, Chuck McCallum, Raman Prasad, Philip Durbin, Sílvia Casacuberta Puig, Ethan Cowan, Vicki Xu, Zachary Ratliff, Nicolás Berrios, Alex Whitworth, Michael Eliot, Christian Lebeda, Oren Renard, and Claire McKay Bowen. OpenDP Library.
[31] Yukiyasu Tsunoo. Cryptanalysis of block ciphers implemented on computers with cache. *preproceedings of ISITA 2002*, 2002.
[32] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.
[33] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. *Cryptology ePrint Archive*, 2018.