
# Orchard: Differentially Private Analytics at Scale

Edo Roth, Hengchu Zhang, Andreas Haeberlen, Benjamin C. Pierce

*University of Pennsylvania*

## Abstract

This paper presents Orchard, a system that can answer queries about sensitive data that is held by millions of user devices, with strong differential privacy guarantees. Orchard combines high accuracy with good scalability, and it uses only a single untrusted party to facilitate the query. Moreover, whereas previous solutions that shared these properties were custom-built for specific queries, Orchard is general and can accept a wide range of queries. Orchard accomplishes this by rewriting queries into a distributed protocol that can be executed efficiently at scale, using cryptographic primitives.

Our prototype of Orchard can execute 14 out of 17 queries chosen from the literature; to our knowledge, no other system can handle more than one of them in this setting. And the costs are moderate: each user device typically needs only a few megabytes of traffic and a few minutes of computation time. Orchard also includes a novel defense against malicious users who attempt to distort the results of a query.

## 1 Introduction

When operating a large distributed system, it is often useful to collect some data from the users' devices—e.g., to train models that will help to improve the system. Since this data is often sensitive, differential privacy [28] is an attractive choice, and several deployed systems are using it today to protect the privacy of their users. For instance, Google is using differential privacy to monitor the Chrome web browser [31], and Apple is using it in iOS and macOS, e.g., to train its models for predictive typing and to identify apps with high energy or memory usage [7, 8]. Other deployments exist, e.g., at Microsoft [27] and at Snap [68].

Today, this data is typically collected using *local* differential privacy [31]: each user device *individually* adds some random noise to its own data and then uploads it to a central entity, which aggregates the uploads and delivers the final result. This can be done efficiently at scale, but the final result contains an enormous amount of noise: as Google notes [14], even in a deployment with a billion users, it is easy to miss signals from a million users. Utility can be improved by reducing the amount of noise, but this weakens the privacy guarantee considerably, to the point where it becomes almost meaningless [79].

One way to avoid this problem is to collect the data using *global* differential privacy instead. In this approach, each device provides its raw, un-noised data to the central aggregator, which then adds random noise only once. This clearly produces results that are more precise, but it also requires a lot of trust in the aggregator, who now receives the individual users' raw data and must be trusted not to look at it. Cryptographic techniques like multiparty computation [83] and fully homomorphic encryption [38] could theoretically avoid this problem, but, at least with current technology, scaling either approach to millions of participants seems implausible.

The recently proposed Honeycrisp system [76] can provide global differential privacy at scale, with a single, untrusted aggregator. Instead of fully homomorphic encryption, Honeycrisp uses additively homomorphic encryption, which is much more efficient. However, the price to pay is that Honeycrisp can answer only one specific query, namely count-mean sketches [8] with additional use of the sparse-vector operator. This query does have important applications (for instance, it is used in Apple's iOS), but it is by no means the *only* query one might wish to ask: the literature is full of other interesting queries that can be performed with global differential privacy (e.g., [15, 31, 40, 41, 55, 64, 70, 82]). Right now, we are not aware of any systems that can answer even one of these queries at scale, using only a single, untrusted aggregator.

In this paper, we show how to substantially expand the variety of queries that can be answered efficiently in this highly distributed setting. Our key insight is that many differentially private queries have a lot more in common than at first meets the eye: while most of them transform, group, or otherwise process the input data in some complicated way, the heart of the algorithm is (almost) always a sequence of sums, each computed over some values that are derived from the users' input data. This happens to be exactly the kind of computation that Honeycrisp's collect-and-test (CaT) primitive can perform efficiently, using additively homomorphic encryption. Thus, CaT turns out to be far more general than it may seem: it can perform the distributed parts of many queries, leaving only a few smaller computations that can safely be done by the aggregator, or locally on each user device.

The key challenge is that, for many queries, the connection to sums over per-user data is far from obvious. Many differentially private queries were designed for a centralized setting where the aggregator has an unencrypted data set and

can perform arbitrary computations on it. Such queries often need to be transformed substantially, and existing operators need to be broken down into their constituents, in order to expose the internal sums. Moreover, a naïve transformation can result in a very large number of sums—often far more than are strictly necessary. Thus, optimizations are needed to maintain efficiency.

We present a system called Orchard that can automatically perform these steps for a large variety of queries. Orchard accepts centralized queries written in an existing query language, transforms them into distributed queries that can be answered at scale, and then executes these queries using a generalization of the CaT mechanism from Honeycrisp. Among 17 queries we collected from the literature, Orchard was able to execute 14; the others are not a good fit for our highly distributed setting and would require a different approach.

Our experimental evaluation of Orchard shows that most queries can be answered efficiently: with 1.3 billion users (roughly the size of Apple's macOS/iOS deployment [6]), most user devices would need only a few megabytes of traffic and a few minutes of computation time, while the aggregator would need about 900 cores to get the answer within one hour. For queries that make use of the sparse-vector operator, this is competitive with Honeycrisp; for the other queries we consider, we are not aware of any other approach that is practical in this setting. In summary, our contributions are:

- the observation that many differentially private queries can be transformed into a sequence of noised sums (Section 2);
- a simple language for writing queries (Section 3);
- a transformation of queries in this language to protocols that can answer them in a distributed setting, using only a single, untrusted aggregator (Section 4);
- the design of Orchard, a platform that can efficiently execute the transformed queries (Section 5);
- a prototype implementation of Orchard (Section 6); and
- an experimental evaluation (Section 7).

We discuss related work in Section 8 and conclude the paper in Section 9.

## 2 Overview

**Scenario:** We consider a scenario—illustrated in Figure 1—with a very large number of users (millions), who each hold some sensitive data, and a central entity, the *aggregator*, that wishes to answer queries about this data. We assume that each user has a device (say, a cell phone or a laptop) that can perform some limited computations, while the aggregator has access to substantial bandwidth and computation power (say, a data center).

**Threat model:** We make the OB+MC assumption from [76]—that is, we assume that the aggregator is honest-but-curious

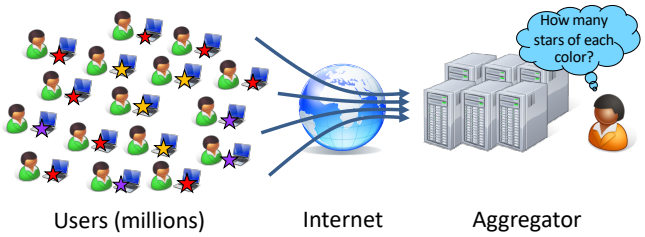


Figure 1: Scenario.

(HbC) when the system is first deployed and usually remains HbC thereafter, but may occasionally be Byzantine (OB) for limited time periods; for instance, the aggregator could be a large company that is under public scrutiny and would not violate privacy systematically, but may have a rogue employee who might tamper with the system and not be discovered immediately. For the users, we assume that most of them are correct (MC) but that a small percentage—say, 2–3%—can be Byzantine at any given time. This is different from the typical assumption in the BFT literature, where one often assumes that up to a third, or even half, of the nodes can be Byzantine. However, BFT systems are typically a lot smaller than the systems we consider: with 4–7 replicas, compromising a third of the systems means just one or two nodes, whereas, in Apple's deployment with 1.3 billion users, a 3% bound would mean 39 million malicious users, which is much larger than, e.g., a typical botnet.

**Assumptions:** Our key assumptions are (1) that the approximate number of users is known and (2) that the adversary cannot create and collude with a nontrivial number of Sybils. For instance, the devices could have hardware support for secure identities, such as Apple's T2 chip or Intel's SGX.

**Goals:** We have four key goals for Orchard:

- **Privacy:** The amount of information that either the aggregator or other users can learn about the private data of an honest user should be bounded, according to the formulation of differential privacy.
- **Correctness:** If all users are honest, the answers to queries should be drawn from a distribution that is centered on the correct answer and has a known shape;
- **Robustness:** Malicious users should not be able to significantly distort the answers; and
- **Efficiency:** Most users should not need to contribute more than a few MB of bandwidth and a few seconds of computation time per query.

### 2.1 Differential privacy

Differential privacy [28] is a property of randomized queries that take a database as input and return an aggregate output. Informally, a query is differentially private if changing any single row in the input database results in "almost no change" in the output. If each row represents the data of a single individual, this means that any single individual has a statistically

| Query | | Support |
|---|---|---|
| Decision-tree learning (ID3) | [34] | Yes |
| k-means | [15] | Yes |
| Perceptron | [15] | Yes |
| Principal Component Analysis (PCA) | [15] | Yes |
| Logistic regression | [2] | Yes |
| Naïve Bayes | [85] | Yes |
| Neural Network training (Grad. Descent) | [2] | Yes |
| Histograms | [82] | Yes |
| k-Medians | [40] | Yes |
| Cumulative Density Functions | [55] | Yes |
| Range queries | [45] | Yes |
| Bloom filters (RAPPOR) | [31] | Yes |
| Count Mean Sketch | [8] | Yes |
| Sparse vector (Honeycrisp) | [76] | Yes |
| Iterative Database Construction | [41] | No |
| Teacher Ensembles (PATE) | [64] | No |
| Vertex programs (DStress) | [63] | No |

Table 1: Selection of differentially private queries from the literature, and support by Orchard.

negligible effect on the output. This guarantee is quantified in the form of a parameter, $\varepsilon$, which controls how much the output can vary based on changes to a single row. Formally, we say that $q$ is $\varepsilon$-*differentially private* if, for any two databases $d_1$ and $d_2$ that differ in a single row, and any set of outputs $R$,

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

In other words, a change in a single row results in at most a multiplicative change of $e^{\varepsilon}$ in the probability of any output, or set of outputs.

A standard method for achieving differential privacy for numeric queries is the *Laplace mechanism* [28], which involves two steps: first calculating the *sensitivity*, $s$, of the query—which is how much the un-noised output can change based on a change to a single row—and second, adding noise drawn from a Laplace distribution with scale parameter $s/\varepsilon$; this results in $\varepsilon$-differential privacy. For queries with discrete values, the standard method is the *exponential mechanism* [56], which defines a "quality score" $q(d,x)$ that measures how well a value $x$ represents a database $d$, and then selects value $x$ with probability proportional to $e^{\frac{\varepsilon q(d,x)}{2s}}$, where $s$ is the sensitivity of $q$. This again results in $\varepsilon$-differential privacy.

Differential privacy is compositional, that is, if we evaluate two queries $q_1$ and $q_2$ that are $\varepsilon_1$- and $\varepsilon_2$-differentially private, respectively, then publishing the results from both queries is at most $(\varepsilon_1 + \varepsilon_2)$-differentially private. This property is often used to keep track of the amount of private information that has already been released: we can define a *privacy budget* $\varepsilon_{\max}$ that corresponds to the maximum loss of privacy that the subjects are willing to accept, and then deduct the "cost" of each subsequent query from this budget until it is exhausted. For a detailed discussion of $\varepsilon_{\max}$, see, e.g., [46].

By now, there is a rich literature on differential privacy proposing many different forms of queries for many different use cases. We have done a careful survey to collect examples that would make sense in our highly distributed setting; Table 1 contains the queries we found, which will also be used in our evaluation (Section 7.1).

## 2.2 Alternative approaches

**Local differential privacy (LDP):** As discussed earlier, another way to avoid trusting the aggregator is to use LDP [31]—that is, for each user to add noise to his or her data individually, *before* uploading it to the aggregator, instead of noising just the final result. However, there are two important challenges. The first is that the noise in the final result now grows with the number of users: for instance, a sum of values from $N$ users now contains $N$ draws from a Laplace distribution $L(\frac{s}{\varepsilon})$, instead of just one! The effective error grows a bit more slowly, with $\Theta(\sqrt{N})$ [29, §12.1], but still, with $N = 10^9$ and $\varepsilon = 0.1$, the median error will be approximately 300,000 with LDP and only 10 with GDP—a difference of several orders of magnitude, which can be severely limiting in practice [14]. The second challenge is that the noise is added by the *users* and not by the aggregator; thus, even a very small number of malicious users can, by using large, correlated values as their "noise" terms, severely distort the final result [22]. We will revisit this problem in Sections 5.3 and 7.3.

**Multiparty computation (MPC):** In principle, the data could also be aggregated using MPC [83], a cryptographic technique that enables a group of participants to jointly evaluate a function $f$ such that each participant only learns the final output of $f$, but not the inputs of each participant. It may seem that all we need to do is set $f := q \circ L(\frac{s}{\varepsilon})$, where $q$ is the query and $L$ is a draw from an appropriate Laplace distribution. The problem, however, is efficiency: generic MPC scales poorly with the number of participants. While there are very efficient solutions for two parties (e.g., [49]) and reasonably efficient ones for a few dozen parties (e.g., [81]), we are not aware of a technique that would be practical with millions or billions of participants.

**Fully homomorphic encryption (FHE):** With FHE [38], users could encrypt their data with a public key and upload them to the aggregator, who could run the query on the ciphertexts, add noise, and then decrypt only the final result using a private key. As with MPC, this approach works for arbitrary queries, and it has the advantage that most of the work is done by the aggregator. However, if the aggregator has the private key, it can also decrypt the users' individual uploads—and even if this problem were solved somehow, computation on FHE ciphertexts is still many orders of magnitude slower than computation on plaintexts, so, with a billion participants, this approach does not seem realistic.

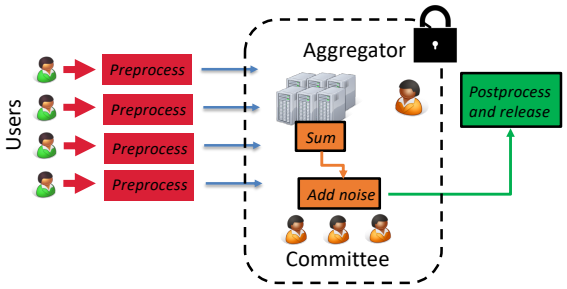

Figure 2: CaT workflow.

## 2.3 Honeycrisp

Honeycrisp [76] can efficiently answer one specific query (namely count-mean sketches) in our setting. As in the hypothetical FHE approach, users encrypt their private data and upload only the ciphertexts to the aggregator; however, there are two critical differences. The first is that Honeycrisp uses *additively* homomorphic encryption, which is orders of magnitude faster than FHE and can be done efficiently at scale. The second is that, to prevent the aggregator from decrypting individual ciphertexts, Honeycrisp delegates key generation and decryption to a small *committee* of 20–40 randomly selected user devices, which uses MPC to perform these (small) tasks. As before, this enables the aggregator to do all of the "heavy lifting" (collecting and aggregating ciphertexts) without ever seeing unencrypted data from individual users; thus, the aggregator does not need to be trusted.

The main drawback of Honeycrisp is that it only supports a single query. Internally, it uses a primitive called Collect-and-Test (CaT), which works roughly as follows (see also Figure 2): each user device computes a vector of numbers, encrypts it with a public key that was generated by the committee, and uploads it to the aggregator, which sums up the ciphertexts using the additive homomorphism. The aggregator then proves to the users that it has computed the sum correctly (which the aggregator, in its Byzantine phases, may not necessarily do); if so, the committee noises and decrypts the final result. This is the primitive that we leverage for Orchard.

Notice that CaT aggregates *vectors*, not just individual numbers. For additively homomorphic encryption, Honeycrisp uses Ring-LWE, which has large ciphertexts that can be subdivided into many smaller fields; these can then be aggregated in parallel. The choices from [76] yield 4,096 counters with about 50 bits each; thus, a single invocation of CaT can efficiently sum up vectors with thousands of elements. We will leverage this fact for our query optimizations (Section 4.5).

## 2.4 Approach and roadmap

Our key insight is that CaT is far more general than it might appear: indeed, the sums it can compute are at the heart of a wide range of differentially private queries. (This is not a coincidence: in fact, a common way to certify differential privacy—e.g., in [10,25,36,42,72,84]—is to use a linear type system to track how much a change in a single user's data can affect a given sum or count.) Thus, by rewriting queries to take advantage of CaT, we can considerably expand the range of queries that can be answered at scale. At a high level, Orchard works as follows:

1. The analyst submits her query as a *centralized* program that computes the desired answer based on a (hypothetical) giant database that contains data from all users. Orchard verifies that the query is differentially private (Section 3).
2. Orchard transforms this program into a distributed computation that relies on CaT, using several optimizations—such as vectorization—to ensure efficiency (Section 4).
3. Orchard executes the distributed program, using protocols from Honeycrisp with some additional steps, and returns the answer to the analyst (Section 5).

# 3 Query language

There are several existing programming languages (e.g., [10, 26, 36, 42, 57, 59, 84, 85]) that can certify differential privacy. Rather than proposing yet another, we adopt an existing language, Fuzz [42]. Fuzz is a functional language, which simplifies our transformations, and its privacy analysis is driven by lightweight type annotations, which is convenient for the analyst. However, the choice is not critical; other languages could be used as well.

## 3.1 Running example: $k$-means

To conserve space, we introduce the Fuzz language through an example: the widely used $k$-means clustering algorithm, shown in Figure 3, which will also be our running example for the rest of this paper. For a more complete description of Fuzz, please see Appendix A.

The $k$-means algorithm divides a given set of points (the input data) into $k$ clusters and returns a centroid for each cluster. It proceeds in several iterations; for clarity, the figure shows only the iteration step, with $k$ hard-coded to 3. The `step` function is given the current estimates of the centroid positions, `c1`, `c2`, and `c3`, and the set of points `pts`; it first assigns each point to the closest centroid, based on the $l_2$ distance (`assign`), and then partitions the set of points into three subsets, one for each centroid. Finally, it produces three new centroid positions `c1'`–`c3'` for the next iteration by averaging the coordinates of the points in each subset. This is done by first summing up the coordinates in each partition, and by counting the points; then the `lap` primitive adds Laplace noise to the sums and counts, and then performs the division.

## 3.2 Language features

In most ways, Fuzz is a conventional functional language; just two special features are relevant here. One is that it has

a *linear type system*, described in [72], that certifies an upper bound on the sensitivity of all operations on private data; when a noising primitive such as `lap` (for the Laplace distribution) or `em` (for the exponential mechanism) is invoked, the parameter $s$ (Section 2.1) is known, and the noise can be drawn from the correct distribution. The other feature is a *probability monad* that ensures that no private data can "escape" from the program without having passed through `lap` or `em` first. Together, these features ensure that, as long as the top-level program has a type of a certain form, it is guaranteed to be differentially private.

Fuzz encapsulates private data in variables of a special type, `bag`, which represents a set with one element for each individual who contributed data. There are several primitives that operate on bags: `bmap` applies a given function to each element of a bag, `bfilter` removes elements for which a given predicate returns false, and `bpartition` splits a bag into several sub-bags, based on the value a given function returns for each element. All of these primitives take bags as arguments and produce new bags, so the private data remains confined in bags. The final bag primitive is `bsum`, which adds up the elements of a bag.

## 3.3 Alternative languages

Using a language other than Fuzz should not be difficult because the key to Orchard, the basic structure of summing followed by a release mechanism, is present in many other languages for differential privacy. Notice that, in Fuzz, summing via `bsum` is the only way to turn bags into data values that can potentially be released. A similar structure is present, e.g., in PINQ [57], which has three aggregation primitives, of which one (`NoisySum`) is equivalent to `bsum` followed by `lap`; the other two (`NoisyAvg` and `NoisyMed`) are equivalent to `bsum` followed by `em`. Another imperative example, Fuzzi [85], supports the addition of new aggregation primitives through an extension mechanism, but the information we need could be specified as part of the extension. The critical features Orchard needs are 1) a sensitivity analysis and 2) a way to recognize the aggregation primitives in the code.

Another possible approach would be to embed Fuzz as a library into a more traditional data analytics language, such as Python3. This embedded-language approach has already seen success in Deep Learning frameworks, such as TensorFlow [1] and PyTorch [66].

# 4 Query transformation

Next, we describe how Orchard transforms centralized Fuzz queries so that they can be executed in a distributed setting.

## 4.1 Program zones

We begin by observing that, if a Fuzz program is differentially private, it necessarily has a very specific structure and can be broken into three different "zones" (which we color-code in our example in Figure 3):

```
assign c1 c2 c3 pt =
  let d1 = sqdist c1 pt
      d2 = sqdist c2 pt
      d3 = sqdist c3 pt
  in if d1<d2 and d1<d3 then 0 else
  if d2<d1 and d2<d3 then 1 else 2

noise totalXY size = do
  let (x, y) = totalXY
  in do x' ← lap 1.0 x
        y' ← lap 1.0 y
        size' ← lap 1.0 size
  return (x'/size', y'/size')

totalCoords pts =
  let ptxs = bmap fst pts
      ptys = bmap snd pts
  in (bsum 1.0 ptxs, bsum 1.0 ptys)

countPoints pts =
  bsum 1.0 (bmap (\pt → 1) pts)

step c1 c2 c3 pts =
  let [p1, p2, p3] =
        bpartition 3 (assign c1 c2 c3) pts
      p1TotalXY = totalCoords p1
      p1Size = countPoints p1
      p2TotalXY = totalCoords p2
      p2Size = countPoints p2
      p3TotalXY = totalCoords p3
      p3Size = countPoints p3
  in do
    c1' ← noise p1TotalXY p1Size
    c2' ← noise p2TotalXY p2Size
    c3' ← noise p3TotalXY p3Size
  return (c1', c2', c3')
```

Figure 3: One step of the $k$-means algorithm, written in Fuzz. The colors represent the "zones" of computation.

- **Red zone** computations run directly on the data of an individual user—here, the `assign` function, which finds the closest centroid for each user's data point.
- **Orange zone** computations are performed on user data that has been aggregated but not yet noised—here, the `lap` operators, which add Laplace noise to the sums.
- **Green zone** computations involve only noised data and constants—here, the final divisions in `noise` and the parts of `iter` that set up the rest of the computation.

The Fuzz type system enforces clear boundaries between these zones: data can only pass from red to orange by aggregation (via `bsum`), and aggregate data can only pass from orange to green by noising (via `lap` or `em`). Moreover, red-zone code always operates on an *individual* element of a `bag`—that is, on data from a a single user. And lastly, none of the operations producing bags offer any way to combine multiple elements of one bag when computing an element of another bag; in other words, every element of every bag that can ever exist is derived (by filtering, partitioning, or mapping) from some single element of some bag that was initially provided as input to the top-level program.

This stratification allows us to map Fuzz programs to Honeycrisp-like computations by mapping the zones to the different parties in Figure 2. Red-zone code is executed directly by user devices; computations in this zone only need the data of one user at a time, so each user device can run it without sending any secrets anywhere. The summation at the red-to-orange boundary can be done as in Honeycrisp, by users encrypting their red-zone outputs and sending them to the aggregator, who adds them up using homomorphic addition and then passes the encrypted sum to the committee. Orange-zone code can be executed by the committee, using MPC, and the members of the committee will be able to decrypt the encrypted sums only after appropriate noise is added. Data that passes from orange to green zones must first pass through a release mechanism (`lap` or `em`) and be thus noised appropriately, so green-zone code can be safely executed "in the clear" by the aggregator itself.

The Orchard compiler uses a special operator to coordinate the mapping, summing, and releasing steps among red, orange and green zones. We call this operator `bmcs` (broadcast, map, clip and sum), and introduce it in the following subsection.

## 4.2 The `bmcs` operator

The operator `bmcs (b,m,c,r)` takes four parameters and behaves as follows:

- first, it *broadcasts* some public state `b` from the aggregator to the user devices;
- on each user device $i$, it *maps* the local private data $d_i$ to a private vector $v_i := \mathtt{m}(\mathtt{b}, d_i)$ using the provided map function `m` (which can use the public state in its computation);
- on each user device, it *clips* the elements of $v_i$ such that $|v_{i,k}| \leq \mathtt{c}_k$; and finally
- it *sums* all these private vectors from all client devices through homomorphic addition to compute $v := \sum_i v_i$ and returns $\mathtt{r}(v)$ using the provided *release function* `r`.

The `bmcs` operator captures the workflow of a single "round" of the distributed protocol; $m$ is the red-zone computation for that round; $r$ is the orange-zone computation. The clipping vector $c$ is needed to guarantee privacy (see Section 5.3).

By rewriting a given Fuzz program to use only `bmcs` rather than the individual bag operations `bmap`, `bfilter`, `bsum`, and `bpartition`, we make its "phase-structure" explicit so that we can directly evaluate it on a Honeycrisp-like distributed platform. We next describe how Orchard does this.

## 4.3 Extracting dependencies

When the analyst submits a Fuzz program to Orchard, Orchard begins by reducing complex bag operations (`bpartition` and `bfilter`) into combinations of the two fundamental bag operations—`bmap` and `bsum`. A `bpartition` that splits a bag into $k$ partitions is reduced into a `bmap` that first maps each value in the bag to a partition index, followed by $k$ `bfilter` operations that filters out each of the individual partitions. A `bfilter` operation is reduced into a `bmap` operation that maps each value $v$ in the bag to an optional value $v'$—when the filter predicate evaluates to true on $v$, the optional value $v' := \mathtt{Some}\, v$, otherwise $v' := \mathtt{None}$.

Orchard then normalizes the program to ensure that all variable names are unique, and that each variable is either the result of a bag operation or the result of a release mechanism (`lap` or `em`). To achieve this, Orchard freshens all variable names, and performs aggressive inlining to eliminate all other variables. Conversely, if a bag operation was originally part of an expression and did not have a name, it is given one. In the resulting normal form, programs make explicit relations between the input database, the intermediate bags and released values, and the output of the program.

Next, Orchard infers dependencies between variables by building a graph with a vertex for each unique program variable. Two vertices $(u, v)$ are connected with a directed and labeled edge $f$ if $v$ is the result of running the bag operation $f$ over $u$. Since the normalized program only contains two simple bag operations, the label $f$ is either the map function supplied to some `bmap`, or the clip bound supplied to some `bsum`. Since Fuzz forbids unbounded loops over private data, this graph is acyclic. Furthermore, since both `bmap` and `bsum` take one bag variable as input and produce another bag variable as output, there is at most one edge between any two vertices in this graph. This implies the graph is in fact a directed tree, and at the root of this tree is the input bag.

This tree is a complete snapshot of the red zone computations encoded in the normalized Fuzz program. Since the dependency tree tells us how to compute any bag value given the bag variable name, we only need to keep bag variable names at their use sites. So we remove all bag operations from the normalized Fuzz program, and use the dependency tree as a reference for emitting code when a bag variable is used. We call the remaining normalized program the "core".

The core contains a mixture of orange zone and green zone computations. Since Orchard eliminates all other program variables in an earlier pass, the variables in the core must either be the result of a bag computation, or the result of a release mechanism. In particular, we call the variables that are results of bag computations "exit vertices" in the tree. (These vertices are scalar numbers, and thus cannot contain any outgoing edges, because no bag operations take scalar numbers as inputs.) By analyzing the core and inspecting the path from the input database to exit vertices, we can emit code in the `bmcs` form.

## 4.4 Transformation to `bmcs` form

The next step traverses the core in a forward pass, while maintaining a intermediate set $S$ of variables. The set $S$ is the

set of variables that are results of release mechanisms at the current program position during the forward pass.

When the traversal encounters a release mechanism (`lap` or `em`), it first compares the set of variables used in this release mechanism against $S$. If the set of used variables is a subset of $S$, then this release mechanism only adds further noise to already released data, and there is no need to invoke `bmcs`.

On the other hand, if a variable $v$ is used in the release mechanism but is is not a member of $S$, then $v$ must be the result of some bag operation. In this case, we must invoke `bmcs` to compute $v$ and release.

Let $p$ be the path from the input database to the variable $v$. Orchard now computes a map function $m_p$ and a clip value $c_p$ as follows. It initializes $m_p := id$ and $c_p := \infty$, then it traverses $p$ starting from the input database. When it encounters a `bmap f`, it updates $m_p := m_p \circ f$; and when it encounters a `bsum c`, it updates $c_p := c$.

In general, a release mechanism may refer to multiple variables $v_1, \ldots, v_i$ that are results of bag operations. For each $v_i$, Orchard walks its corresponding path $p_i$ to compute $m_{p_i}$ and $c_{p_i}$. It then fuses these map functions and clip bounds into a new map function $m\,db = (m_{p_1}\,db, \ldots, m_{p_i}\,db)$ and a new clip bound $c = c_{p_1} +\!+ \ldots +\!+ c_{p_i}$, where $+\!+$ represents vector concatenation.

Finally, if $f(v_1, \ldots, v_i)$ is the release mechanism that uses program variables $v_1, \ldots, v_i$, we build the release function $r\,sum = f(prj_1\,sum, \ldots, prj_i\,sum)$. Here, *sum* is the aggregated vector, and each $prj_i$ projects the corresponding value for $v_i$ out of the aggregated vector sum.

## 4.5 Optimizations

The transformation process that has been described so far will calculate the correct result, but in general it will produce many redundant `bmcs` operations because it walks the core in a forward pass and emits one `bmcs` call for each release mechanism that uses private data. We can do better by observing that release mechanism calls often do not depend on each other (such as the three calls to `noise` in the k-means example) and can in fact be fused into one `bmcs` call.

Orchard exposes these optimization opportunities to the code transformation process through a simple source code rewriting step. After Orchard has inlined and normalized the input Fuzz program, but before code transformation into `bmcs`, Orchard performs local dependency analysis on release mechanism calls, using a marker combinator `par` to combine release mechanisms that have no dependency relations.

For example, the three `lap` calls in the `noise` function for the kmeans example will be rewritten into:

```
((x', y'), size') ←
  par (par (lap 1.0 x) (lap 1.0 y))
      (lap 1.0 size)
```

Since Orchard inlines the `noise` function, in fact all nine `lap` calls in the `step` function for the *k*-means example will be combined through the marker `par` combinator (there are three `lap` calls in each `noise` call, and there are three `noise` calls).

The purpose of the `par` combinator is to allow code transformation to fuse release mechanisms together just by looking at the syntax of the program under analysis. In the last phase of code transformation, when Orchard encounters a `par` combinator, it first recursively emits the map and release functions for the two arguments to `par`. Let us call these map functions $m_1$ and $m_2$, and the release functions $r_1$ and $r_2$. Next, Orchard fuses them together by creating a new map function $m\,db = (m_1\,db, m_2\,db)$, and a new release function $r\,sum = (r_1\,sum, r2\,sum)$. The clip bounds are concatenated to produce a fused clip bound. The code transformation recursively fuses the release mechanisms combined with nested `par` combinators, until finally only a single `bmcs` call is emitted for all of the combined release mechanisms.

## 4.6 Limitations

Our implementation currently insists that all loops in the red and orange zones terminate after a finite number of rounds, and it disallows unbounded recursion in these zones. Finite loop bounds are common in the differential privacy literature because they simplify the reasoning about the privacy cost; queries with unbounded loops, such as the PrivTree algorithm [86], tend to require more sophisticated reasoning, and thus cannot be verified by most automatic checkers. If necessary, the limit in the red zone could be replaced with timeouts and default values [42]. Notice that we *do* allow unbounded loops in the green zone, so we can still use dynamic predicates to check for convergence, e.g., in k-means clustering.

Orchard's front end relies on an existing programming language and type system, and it inherits their limitations. In particular, if a query is differentially private but the Fuzz type system cannot prove it, Orchard will reject it, and if a query's real sensitivity is $s_1$ but Fuzz only derives a sensitivity value $s_2 > s_1$, Orchard will use $s_2$. These limitations could be removed by using a different source language – e.g., one with a more advanced type system, such as DFuzz [36], or one that allows the analyst to help with the privacy proofs, such as apRHL [4].

Orchard's optimization for fusing independent release mechanisms only recognizes fusion opportunities for release mechanisms that are syntactically next to each other. Due to this simplistic nature, Orchard may miss opportunities for fusion of release mechanisms that are only revealed through a more global dependency analysis. However, in our experiments, we find that this limitation does not prevent us from emitting code with the optimal number of `bmcs` calls. We plan on improving the fusion analysis in future work.

# 5 Query execution

Next, we describe the platform Orchard uses to execute distributed queries once they have been transformed using the method from the previous section.

## 5.1 Overall workflow

Orchard implements `bmcs` using the CaT primitive from Honeycrisp [76], with three important additions: Orchard supports more than one round, it adds the broadcast step (which was not needed for Honeycrisp's one hard-coded query), and it supports more general computations on the user devices and within the committee's MPC (which Orchard needs for the red and orange zones). Protocols for sortition and verifiable aggregation (discussed below) are used verbatim, so the correctness proofs from [76] still apply. The platform consists of two components: a *server*, which runs in the aggregator's data center, and a *client*, which runs on each user's device (e.g., phone or laptop). These components operate as follows.

**Setup:** When an analyst wants to ask a query, she formulates it in the language from Section 3 and submits it to the server. The server typechecks the query, to verify that it is differentially private; if not, it aborts. The server then transforms the query as described in Section 4, but keeps only the code for the green zone. The server then triggers a *sortition* protocol that causes a very small, random *committee* of user devices to be elected. (As in Honeycrisp, a typical committee size is about 30–40, out of perhaps $10^9$ devices.) The server sends the query to the committee, whose members perform the same transformation as the server but keep only the code for the orange zone of each `bmcs` operation, as well as the associated privacy costs $\varepsilon_i$. The committee runs an MPC to generate a keypair for an additively homomorphic cryptosystem, and each committee member keeps a share of the private key. The server then executes the prefix (if any) of the green-zone computation that does not involve private data.

**Broadcast:** When the server encounters the $i$th `bmcs` operation, it sends the sequence number $i$ to the committee. The committee deducts $\varepsilon_i$ from the privacy budget $\varepsilon_{\text{max}}$ and, if this succeeds, signs an *execution certificate* that contains the query, the public key, and the sequence number $i$ of the `bmcs`, and returns the certificate to the server. This certificate is needed to convince the clients that the server has "paid" the privacy cost $\varepsilon_i$ for the specific step they are about to execute; the sequence number prevents query reexecution without charging the privacy budget again.

**Map and clip:** The server now distributes the certificate, along with any broadcast state in the `bmcs`, to the clients. Each client (1) verifies that the committee was elected properly, that the execution certificate is signed by the committee, and that the certificate is not a duplicate; (2) transforms the query to obtain the red-zone computation for the $i^{th}$ `bmcs` operation; (3) executes the red-zone code on its local data; (4) encrypts the result with the public key from the certificate; and (5) uploads the result to the server, along with a zero-knowledge proof that (a) the local input was in the correct range; (b) the red zone was executed correctly; and, if $i > 1$, that (c) the client has not changed its local input since the first `bmcs` in the current query.

**Sum:** The server aggregates all the uploads using homomorphic addition and then publishes a Honeycrisp-style summation tree, so the clients can verify that it has included each user's data exactly once; if not, they can report the aggregator. Next, the committee performs another MPC to execute the orange-zone code (which noises and decrypts the computed aggregate) and then sends the plain-text result to the server, which uses it as the result of the `bmcs` operation and continues executing the green-zone code. If the server encounters further `bmcs` operations, it repeats the broadcast, map, clip, and sum steps for each of them.

## 5.2 Security: Aggregator

One key difference from Honeycrisp is that Orchard's red- and orange-zone computations are not hard-coded and must be compiled from the query instead. A naïve approach could have been to have only the server perform the transformation and to have it provide the red- and orange-zone code to the committee and to the clients, respectively. However, in this case it would have been easy for the server to, say, replace the orange zone with the identity function (to disable noising) and/or to replace the red zone with "if the user is Alice, return data $\times 10^9$, else 0" (without proper clipping).

Orchard avoids this issue by (1) having the committee and the clients compile the red and orange zones directly from the original query and by (2) including the query in the execution certificate, so that all correct participants can be sure they are part of the *same* query. Since a correct client or committee member would perform the compilation as specified, it would (correctly) reject any proposed query that was not differentially private, and it would include all the necessary elements, such as clipping and noising. A dishonest server still has control over the green zone and can run any arbitrary code there. However, it can only hurt itself by doing this: the users' privacy is guaranteed by the red and orange zones, and any data that reaches the green zone is already properly declassified.

Of course, the aggregator can misbehave in several other ways, but the compilation attack is the only one that is specific to Orchard; the others were already possible in Honeycrisp, and the defenses from Honeycrisp continue to apply. For completeness, we briefly review some key defenses below; for a complete description, please see [76, §3].

**Privacy budget:** A malicious aggregator could try to run more queries than the privacy budget allows. To prevent this, the budget balance is maintained by the committee. In each round, the committee checks whether the remaining privacy budget is sufficient to execute the query; if so, it signs a query

authorization certificate that includes, among other things, the remaining budget and the current round number. This certificate is sent to all user devices, which check it before uploading their responses. If the committee changes, the new members rely on the budget from the previous round's certificate.

**Targeting individuals:** A malicious aggregator could try to learn the private data of specific users by performing the aggregation incorrectly – perhaps by leaving out data from certain users, or by multiplying the encrypted data from other users with a large constant (which is possible in an additively homomorphic cryptosystem), or even by pretending that a single user's data is the result of the entire aggregation. To prevent this, Orchard requires the aggregator to construct a *summation tree* to prove that it has computed the aggregation correctly. Each user device checks a small portion of this tree.

**Reporting channel:** We assume that there is an external channel that devices can use to report the aggregator, if they should discover that the aggregator has misbehaved. Like Honeycrisp, Orchard produces evidence that the devices can use to substantiate such a report; for instance, this evidence could be posted in an online forum (Twitter, Wikipedia, ...) or it could be given to the press. In a large-scale deployment, the aggregator would typically be a large entity with a reputation to lose, so this mechanism should provide an incentive for the aggregator to follow the protocol correctly.

**Collusion:** If the aggregator is also the manufacturer of the user devices (which would be the case, e.g., in a deployment by Apple or Google), a malicious aggregator could try to roll out a backdoored OS version or manufacture a large number of additional devices, with which it could then collude. Here, our assumption that the aggregator is Byzantine only *occasionally* (the OB in our OB+MC assumption) is critical, because it limits the potential impact of such misbehavior.

**Committee tampering:** For a committee of size C, Orchard requires that $\frac{2C}{5}$ committee members are honest. With 2–3% Byzantine users, as we have assumed in Section 2, the chances of randomly sampling a committee with too many Byzantine users are miniscule; with $C = 40$, the chances of ever encountering it during a period of ten years, with one round every day, would be about 0.001%. However, a malicious aggregator could try to increase this probability by preventing honest users from participating in the sortition. To defend against this, the aggregator must maintain a Merkle tree of all the users, so that the results of the election are verifiable by all devices.

## 5.3 Security: Malicious clients

Another key difference from Honeycrisp is that there can be more than one `bmcs` invocation and that clients can potentially learn some information about the result of previous invocations from the broadcast step. This is not a privacy issue because the type system ensures that any broadcast state has been properly noised, but a group of malicious clients could potentially use this information in a targeted attack.

As a concrete example, suppose a large online retailer uses the $k$-means algorithm from Figure 3 to calculate the positions for $k$ new shipping centers, based on the locations of their current customers; suppose, further, that a small group of users wishes to ensure that one of the centers is built in their home town. Notice that each `bmcs` broadcasts the set of centroids from the previous round. In the last round, the attackers can use this information to calculate exactly (modulo noise) what their locations would need to be to move the nearest centroid to their town and then change their inputs accordingly.

To prevent adaptive attacks like this, Orchard can optionally use verifiable computation (VC) [65] on the client side. When this is enabled, clients must upload a cryptographic commitment to their local data along with their first `bmcs` response, and they must include, with each response, a zero-knowledge proof that (a) they have executed the red-zone code correctly and (b) their initial commitment opens to the input they used in the current round. With this defense, the attackers can only choose their initial inputs. As we will show in Section 7.3, this makes a successful attack much harder.

## 5.4 Handling churn

A third difference is that Orchard computations with multiple `bmcs` rounds can take much longer than Honeycrisp's single-round computation. This raises two concerns: (1) the workload of the committee is somewhat higher, and (2) devices are more likely to go offline during the computation.

To address the first concern, Orchard can optionally choose a fresh committee after a few `bmcs` rounds. This requires a few more devices to serve on committees, and it adds a bit more work for the overall system because each new committee has to generate a fresh keypair, but it is safe, and it limits the work that any given committee member has to perform. To address churn in the committee, Orchard uses Shamir secret sharing to ensure that the committee can reconstruct the private key even if it has lost a few of the shares because the corresponding committee members have gone offline.

This leaves the concern that some *user* devices will leave (and others join) between rounds. This does not affect correctness, since the red zone retains no state between rounds, but it does mean that the `bmcs` sums could be computed over data from slightly different sets of users. Almost by definition, differential privacy cannot release anything that is specific to particular users, so the overall impact of individual user arrivals or departures should be small [29, §2.3.2]. The effect of higher levels of churn depends on the algorithm and on the kinds of users that are joining or leaving. For instance, consider the effect that a major power outage in a large geographic region – say, the 2003 blackout in the Northeastern U.S. [33] – would have on a query that was already in progress. If the query was choosing facility locations within the United

States, the results would be severely distorted, since it would suddenly appear as if there were no users in the Northeast at all. If, however, the query was measuring the age distribution of the users, the impact would be small, since the age distribution in the Northeast would be roughly comparable to the age distribution elsewhere.

# 6 Implementation

For our experiments, we built a prototype of Orchard. We used Haskell to implement the Fuzz frontend and the transformations, and Python for the backend. Our prototype generates and runs the actual red-zone and orange-zone code; for the aggregation (which would be done with millions of users in a real deployment), we benchmark the individual steps and then extrapolate the cost. Overall, our prototype consists of about 10,000 lines of code, and is publicly available [62].

**Encryption:** For additively homomorphic encryption, we use the Ring-LWE scheme [54]. This works over a polynomial ring $R_p := \mathbb{Z}_p[x]/(x^n+1)$, where $p$ is a prime and $n$ is a power of 2. The secret key is a random polynomial $s(x) \in R_p$, and the public key is a pair generated by sampling a random $a \in R_p$ and setting the public key to be $(a,b) \in R_p^2$, where $b := a \cdot s + e \in R_p$, for some "error" $e \in R_p$ chosen from an appropriate error distribution. The plaintext space is $\mathbb{Z}_q^l$, where $q, l \in \mathbb{Z}$, $l \leq n$, $q \ll p$ and $|p \mod q| \ll q$. To encrypt a vector $z \in \mathbb{Z}_q^l$, the encryptor generates a random $r \in R_p$, and computes the ciphertext $(u,v) := (a \cdot r + e_1, b \cdot r + \lfloor p/q \rceil \cdot z) \in R_p^2$. Decryption is then simply $z = round(v - u \cdot s, \lfloor p/q \rceil) / \lfloor p/q \rceil$, where $round(x,y)$ rounds each coefficient of $x$ to the nearest multiple of $y$. (We assume the errors $e$, $e_1$, $e_2$ are sufficiently small relative to $p/q$.)

This encryption scheme allows us to represent our key generation and decryption protocols with a small constant number of additions and one multiplication in the polynomial ring. Moreover, it allows us to pack many 'slots' of ciphertexts into one large ciphertext, with almost no additional cost. Given our security parameter choices, this scheme yields up to 4,096 counters, each with a capacity of roughly 50 bits.

**MPC:** We use the SCALE-MAMBA framework [50] to implement the MPC operations for key generation and for the orange zones (Section 5.1). For key generation and decryption we used code we obtained from the authors of [76]. SCALE-MAMBA supports an arbitrary number of parties and is secure in the fully-malicious model. Operations are performed in a finite field modulo a configurable prime $p$, which allows for the support of both integers and floating points. This is a natural fit for our Ring-LWE encryption scheme, which also requires an integer modulus, and thus no additional modular arithmetic needs to be implemented within the MPC. In Ring-LWE, the additive homomorphism of plaintexts is modulo some integer $q$, where $|p \mod q| \ll q$; ideally, $p = 1 \mod q$.

**Secret sharing:** SCALE-MAMBA also supports Shamir secret sharing [77]. We use this to shard the private key among the $k$ committee members in such a way that any subset of $t+1$ members can reconstruct the entire key. At the same time, $t$ dishonest nodes cannot learn anything about the key, and $t+1$ honest nodes can detect any errors introduced by dishonest nodes. This enables Orchard to tolerate the loss of a few committee members. We modified the open-source SCALE-MAMBA source code to reconstruct the secret key automatically, if needed, using the remaining shares.

**Verifiable computation:** We use the zk-SNARK protocol [11] to enable clients to prove, in zero knowledge, that they have done the red-zone computation correctly, with consistent inputs (Section 5.3). For benchmarking, we used the implementation from the Pequin toolchain [67].

**Security parameters:** We use the LWE-estimator tool [53] of Albrecht et al. [5] to obtain concrete parameters that provide sufficient security based on the best known attacks on LWE. We chose dimensionality $n = 4096$, a 128-bit prime $p$, and a Gaussian error distribution with $\sigma = \frac{\sqrt{2}}{2}$ (which we approximate as the centered binomial distribution with $N = 2$ trials) in each dimension, which gives over 128 bits of security. For the verifiable aggregation, we use the same choices as Honeycrisp, namely SHA-256 hashes and RSA-2048 signatures.

# 7 Evaluation

Our experimental evaluation is designed to answer four high-level questions: (1) How many private queries can Orchard support? (2) How well do Orchard's optimizations work? (3) How effective are Orchard's defenses against malicious clients? And (4) what are the costs of Orchard?

## 7.1 Coverage

To get a sense of how many (private) queries Orchard can support, we did a careful survey of the differential privacy literature to find queries that are plausible candidates for our highly distributed setting. We collected as many different kinds of queries we could find; we excluded only a) queries that were substantially similar to ones we already had (e.g., different variants of computing CDFs), and b) queries where we simply could not imagine the data being distributed across lots of individual devices.

Table 1 (in the Overview section) shows the queries we found, as well as the papers we found them in. We then implemented each query in Fuzz, taking care to write the queries as they were presented in the papers, and not in a way that would be convenient for Orchard (e.g., with computations already grouped the way `bmcs` would require them).

We found that, out of the 17 queries we found, 14 (82%) were accepted by Orchard. The three queries that did not work were PATE [64], IDC [41], and DStress [63]. These

| Query | Naïve | Optimized |
|---|---|---|
| ID3 | $2md$ | $m+1$ |
| k-means | $3m$ | $m+1$ |
| Perceptron | $2md$ | $m+1$ |
| PCA | $d^2+d$ | 1 |
| Logistic regression | $d+1$ | 2 |
| Naïve Bayes | $2d$ | 2 |
| Neural Network | $2m(d+1)$ | $m+1$ |
| Histograms | $b$ | 1 |
| k-Medians | $3m$ | $m$ |
| CDF | $b$ | 1 |
| Range queries | $b$ | 1 |
| Bloom filters | $d$ | 1 |
| Count Mean Sketch | $d$ | 1 |
| Sparse vector | 1 | 1 |

Table 2: `bmcs` rounds needed for each query, with and without optimizations. $d$ is the input vector length, $m$ the number of iterations, and $b$ the number of buckets (see Section 7.4).

queries are not a good fit for our model. DStress operates on graphs, whereas we assume a set of per-user records. IDC is a "template algorithm" with an oracle function $U$, and good choices for $U$ require functions beyond simple bag operations. PATE requires training private (un-noised) "teacher" models and then training a "student" model with noisy labels provided by the teachers. In our model, only the aggregator could play the role of PATE's teachers, but we do not trust it to see sensitive data in the clear, so we cannot express this algorithm.

Overall, our data suggests that Orchard is able to execute a wide variety of differentially private queries—even though these queries were designed for the centralized model.

## 7.2 Optimizations

A naïve translation of a centralized query typically results in a lot more `bmcs` invocations than necessary. To estimate how much our optimizations can help with this, we compiled each query twice, once with the full transformation and once with optimizations disabled; we then counted the `bmcs` operations in the resulting programs.

Table 2 shows our results. In most cases, our optimizations substantially reduced the number of `bmcs` rounds that were needed. (The exact reduction depends on the parameters.) Since the rounds are done sequentially (the `bmcs` calls in the green-zone code are "blocking"), and since `bmcs` accounts for almost all of a typical query's runtime, this means a much lower processing time.

We manually inspected the optimized code, looking for opportunities to further reduce the number of rounds, but could not find any. In principle, Orchard's optimizations could miss opportunities for fusing release mechanisms (Section 4.6), but this did not occur for any of the queries we tried.

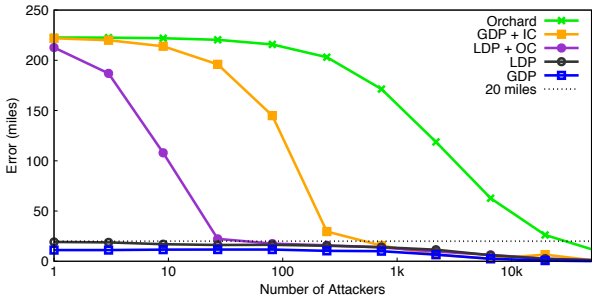


Figure 4: Impact of malicious users.

## 7.3 Robustness to malicious users

To examine how much Orchard's defenses help against malicious users, we implemented the attack scenario from Section 5.3. Recall that this involves an online retailer using $k$-means to find locations for $k = 3$ new shipping centers and a group of attackers trying to cause one of the centers to be built in their home town. We randomly sampled latitudes and longitudes for $N = 10^4$ honest users from a rectangle that includes the lower 48 U.S. states, and we used Seattle, Houston, and New York as reasonable guesses to initialize the centroid positions. We then simulated the behavior of Orchard, as well as four hypothetical alternatives: (1) local differential privacy (LDP); (2) global differential privacy (GDP) with a trusted aggregator; (3) GDP with input clipping (IC), which rejects coordinates outside the valid range and was implemented in [76]; and (4) LDP with output clipping (OC), which requires users to clip their *noised* values to $10\times$ the valid range. The attackers try to move the East Coast centroid (which is near Richmond, VA without the attack) to Pittsburgh, PA, using the strategy from Section 5.3; we assume that the attackers do not have knowledge of any data from previous Orchard queries (because, if this information was still relevant, the aggregator would likely have no need to issue a new query). We vary the number of attackers $A$, and we assume that the attackers are able to estimate $N$ but do not know the locations of the other users. We say that the attack succeeds if the final East Coast centroid is within 20 miles of Pittsburgh.

Figure 4 shows the distance from Pittsburgh of the resulting East Coast centroid for each scenario and with various values for the parameters; the figure shows medians across 500 independent runs. Without a defense, GDP and LDP succumb to even a single attacker, who can observe the centroid's location in the penultimate round and then calculate an input (far outside the valid range) that will move the centroid to Pittsburgh in the final round. The residual error is due to noising; it decreases as $A$ increases. Notice that GDP's error is even lower than LDP's; this is because GDP adds less noise.

With OC, the attackers can no longer report arbitrary values and must instead choose the largest value in the right direction that will be accepted, but the attack still succeeds with about $A = 31$ (0.3% of the users). IC further restricts the range; success now requires $A = 500$ attackers. With Orchard, the

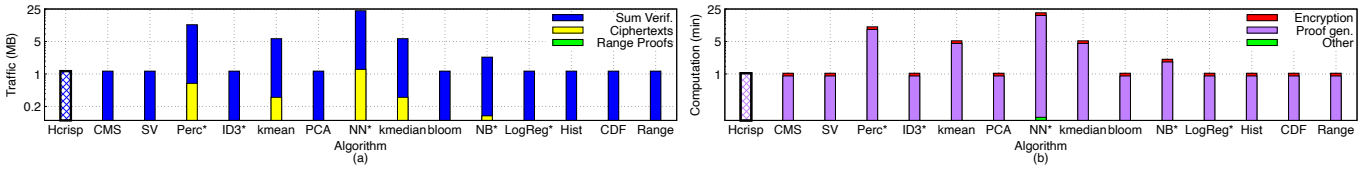


Figure 5: Bandwidth (a) and computation (b) required of each participant in a run of each algorithm.

attackers cannot adapt, and since they do not know up front what values to report—reporting, say, Portland, ME, would risk "overshooting" and moving the centroid away from Pittsburgh again—their best strategy is to simply report Pittsburgh as their location. With this strategy, the attack takes about $A = 20,000$—far more than the number of honest users.

## 7.4 Experimental setup

Next, we used our prototype to measure Orchard's costs to users, committee members, and the aggregator. We benchmarked the client-side software on a laptop with a 2.3 GHz dual-core processor and 8 GB of RAM running macOS Catalina. To simulate committee members operating in a global setting, we used `t2.large` EC2 instances with 8 GB of RAM, located in all available geographic regions (including the U.S., Europe, Asia, and Brazil), to get realistic latencies. For our aggregator experiments we used eight PowerEdge R430 servers with 64 GB of RAM, two Xeon E5-2620 CPUs, and 10 Gbps Ethernet; the operating system was Fedora Core 26 with a Linux 4.3.15 kernel. This equipment seems reasonably close to what a real-world aggregator might have available in its data center.

Many of our algorithms have parameters that affect the cost. For $k$-means and $k$-medians, we chose $m = 5$ and $k = 3$, because [9] notes that, given proper cluster initialization, the solution after five rounds is consistently as good or better than that found by any other method. For Perceptron, we chose $m = 10$, because the algorithm is guaranteed to converge after at most $O(1/\alpha^2)$ iterations, where $\alpha$ is the margin in a linearly separable dataset [75]. With vectors of size 10, we assume 1-separability to get this guarantee. For ID3, we set vector dimension $d = 100$ because we can support estimating entropy for counters of up to vectors of size 1 million (e.g., all possible 6-digit zip codes) with far fewer counters on the aggregator's side. For the neural network, we chose $m = 20$ epochs, for which [44] shows accuracy competitive with SGD.

Since Orchard is a generalization of Honeycrisp, we report Honeycrisp's numbers for comparison. We got these numbers by executing Honeycrisp's fixed query, which compiles to a single `bmcs`, with Orchard's additions disabled.

## 7.5 Cost for normal participants

The key costs to a normal Orchard participant are: (1) the red-zone computation itself; (2) encrypting the value to be uploaded; (3) generating the zero-knowledge proofs; and (4) verifying the aggregator's summation. (The transformations themselves are cheap; this step never took more than 410 ms for any of our 14 queries.) To quantify these costs, we benchmarked the Orchard client while it was executing each of our 14 queries; to get realistic numbers for sum verification, we emulated a system with $N = 1.3 \cdot 10^9$ users for the client to interact with. We measured the number of bytes sent, as well as the computation time spent on Orchard operations.

Figure 5 shows our results. Both the bandwidth and the computation time vary significantly between queries, but they are largely proportional to the number of `bmcs` rounds, whose cryptographic operations dominate the cost. The red-zone computations themselves are typically trivial (many simply return a value), so their cost is very small in comparison; we simply include it with the other protocol overheads in Figure 5(b). Overall, the bandwidth costs are modest, ranging from 1 MB to about 25 MB per query. The computation typically takes at most a few minutes.

The neural-network query is a an outlier; it takes about 25 minutes of computation time, which raises some concerns, e.g., about battery life on mobile devices. This high cost is mostly due to the high number of rounds we used ($m = 20$), to show what would happen when training on a "hard" problem. For "easy" lower-dimensional problems, even a single pass can be statistically optimal [69].

To measure the cost of the defense from Section 5.3, we selectively disabled the part of the zero-knowledge proof that concerns input consistency; this typically reduced the proving time by about 3%. This is because the client already has to prove that the encrypted value is in the correct range; the marginal cost of this extra proof obligation is very small.

## 7.6 Cost for the committee

For each query, Orchard selects a small committee of $C$ user devices that are expected to participate in the key-generation MPC, as well as in the per-`bmcs` MPC that performs decryption and orange-zone computations. To quantify the cost to committee members, we set up committees with EC2 instances as described in Section 7.4, triggered each of our 14 queries, and measured the bandwidth and computation that the two MPCs consume. We report the cost of a single iteration of each MPC.

Figure 6 shows our results; where queries use two `bmcs` rounds per iteration, we report the cost of the more expensive one (indicated with an asterisk). The cost of the key-generation MPC depends only on the key length, and is thus identical for all queries; the cost of the orange-zone MPC

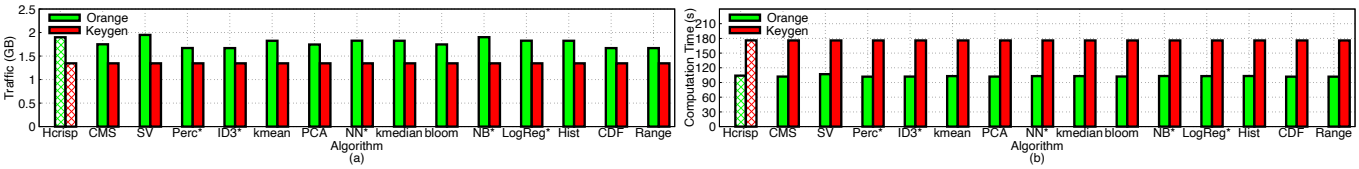


Figure 6: Bandwidth (a) and computation (b) required of each committee member during one round of orange-zone computation.

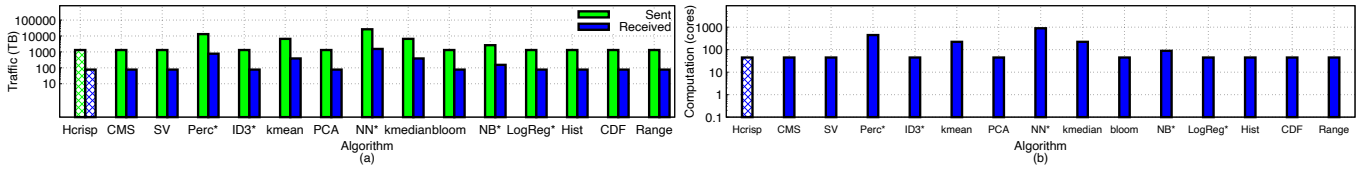


Figure 7: Bandwidth (a) and computation (b) required of the aggregator.

varies with the query, but not by much. Overall, decryption dominates the costs, and, since every `bmcs` call fits into one large packed ciphertext, we see the same behavior for all queries. In absolute terms, these costs are significant; a typical query with one round of `bmcs` consumes about 3 GB of traffic and five minutes of computation time; the total is higher if additional rounds are required.

Notice that the chances of actually being selected for the committee are tiny: for $N = 1.3 \cdot 10^9$ users, a typical committee size is about $C = 40$, so each user is only about $9\times$ more likely to be chosen than to win the jackpot in Powerball. Nevertheless, it may be useful to excuse resource-limited devices, such as mobile phones, from committee service and to rely mostly on devices like desktops and laptops, when possible.

## 7.7 Cost for the aggregator

Next, we quantify the costs of the aggregator, who must collect the input from each device, verify the zero-knowledge proofs, sum up the inputs, generate the summation proof, and distribute this proof to each device. We do not currently have a large enough deployment of Orchard to run this experiment end-to-end, so we estimate the costs based on benchmarks of the individual steps. We set the number of rounds as discussed in Section 7.4, and we report results for $N = 1.3 \cdot 10^9$.

Figure 7 shows the number of bytes the aggregator would need to send for each query, as well as the number of Xeon E5-2620 cores it would need to ensure that the computations do not last for more than one hour. As before, the costs depend mostly on the number of rounds; the cost of the green-zone computation is insignificant. The most expensive query (Neural Network) would require 892 cores, or 74 machines with two E5-2620 CPUs each. It would also involve sending 13,180 TB, which is a lot but actually corresponds to about 10 MB per user. For comparison: the average transfer size of a web page is about 2 MB [47]; typically, much of this is offloaded to CDNs, and the same would be possible for Orchard's summation proofs.

**Scalability:** We also ask how well Orchard scales with the number of participating users $N$. This is mostly a concern for the aggregator: the size of the MPCs (and, thus, the cost for committee members) does not depend on $N$ at all, and the cost for individual users grows only very slowly, with $O(\log N)$, because of the summation trees. We estimate the costs of the aggregator as above, but this time we vary $N$.

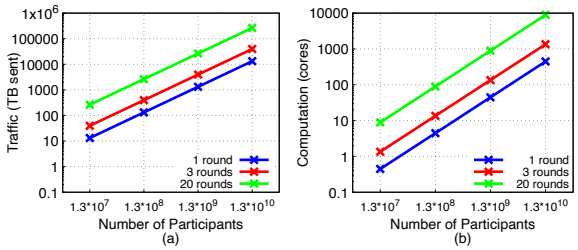


Figure 8: Bandwidth (a) and computation (b) required of the aggregator, for different system sizes.

Figure 8 shows our results (all scales are logarithmic). Although the scaling is technically $O(N \log N)$ because the height of the summation trees grows with $N$ and each user must be sent some paths in the tree for verification, the non-linear component is small in both figures, which means that Orchard scales very well with $N$. This is expected, since Orchard is based on Honeycrisp, which scales similarly, and nothing in Orchard destroys this scalability.

# 8 Related work

To our knowledge, Orchard is the first *general* system that can process a wide variety of queries with (1) a single untrusted aggregator, (2) global differential privacy, and (3) scalability to millions of users.

**Different trust assumptions:** Several other systems require at least some trust in additional parties. Prochlo [14] anonymizes the user data using a shuffler, who must not collude with the aggregator; this reduces the privacy cost of LDP algorithms considerably [30]. Similarly, the crypto service provider in [37, 60] must not collude with the evaluator, and the proxy in PDDP [21] and the aggregator in Leontiadis et al. [51] must not collude with the analyst. UnLynx [35] and Prio [23] use the anytrust model, that is, a group of servers

of which at least one must be honest; SecureML [58] uses a pair of non-colluding servers; and other solutions, such as [20, 24, 48, 74], use a trusted third party for at least some steps. These additional trust assumptions yield substantial benefits, but recruiting parties that will help the aggregator but are sufficiently trustworthy to users may not be easy.

Some solutions, such as [52] use trusted hardware like Intel's SGX. We avoid this approach in Orchard because current TEE implementations are not yet sufficiently trustworthy, as shown, e.g., by the many successful attacks on SGX [61].

**Local differential privacy:** Google's RAPPOR [31, 32] uses LDP to aggregate data; similar systems have been deployed, e.g., by Apple [8], Microsoft [27], and Snap [68]. As discussed in Section 2.2, LDP requires significantly more noise than GDP, which can be limiting in practice [14], and it is vulnerable to attacks from small groups of colluding users [19, 22].

**Smaller scale:** A variety of solutions are available for systems with at most a few thousand users. For instance, Shi et al. [78] use a distributed key generation scheme to remove trust in the aggregator, and [3] use pairwise blinding instead of encryption, but these approaches do not work well under churn. Some systems have scaled MPC to impressive sizes – for instance, SEPIA [18] handles hundreds of users, and Reyzin et al. [73] perform secure aggregation for thousands, by adding homomorphic threshold encryption – but supporting millions of users with MPC seems unrealistic. Bonawitz et al. [17] use secret sharing, but, with $n$ users, several costs grow with $O(n^2)$; Bindschaedler et al. [13] and Goryczka and Xiong [39] require $O(n^2)$ communication; Rastogi and Nath [71] use $(t,n)$-threshold encryption; and Halevi et al. [43] have $O(n)$ latency, since users must interact with the aggregator sequentially.

**Federated learning:** FL [12, 16] is another approach to working with highly distributed data. Most existing systems do not guarantee differential privacy, and the ones that do typically rely on LDP, such as [2]. Zhu at al. [87] recently proposed an interactive protocol with better privacy, specifically for discovering heavy hitters, but it does trust the aggregator with one simple task (thresholding). Truex et al. [80] relies on threshold Paillier, but it is limited to small deployments.

## 9 Conclusion

Prior to Orchard, it may have seemed that running differentially private queries at scale required either making compromises (on privacy, accuracy, or trust) or custom-building a cryptographic protocol. Orchard shows that, because of structural similarities among many queries, general solutions do exist, even when there is only a single, untrusted aggregator. There are still types of queries that Orchard does not support—one interesting example are queries on graphs—but we speculate that, by finding and exploiting similar structural patterns, solutions could be built for some of them as well.

## Acknowledgments

We thank our shepherd Bryan Parno and the anonymous reviewers for their thoughtful comments and suggestions. This work was supported in part by NSF grants CNS-1955670, CNS-1733794, CNS-1703936, CNS-1563873, and CNS-1513694, as well as by a Google Faculty Research Award.

## References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Whitepaper; software available from tensorflow.org.

[2] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proc. CCS*, 2016.

[3] G. Ács and C. Castelluccia. I have a dream! (DiffeRentially privatE smArt Metering). In *Proc. International Conference on Information Hiding (IH)*, 2011.

[4] A. Albarghouthi and J. Hsu. Synthesizing coupling proofs of differential privacy. *Proc. POPL*, 2017.

[5] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptography*, 9:169–203, 2015.

[6] Apple. Apple reports first quarter results. Press release, February 2018; `https://www.apple.com/newsroom/2018/02/apple-reports-first-quarter-results/`.

[7] Apple. Differential privacy. `https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf`.

[8] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 1(8), Dec. 2017.

[9] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. In *Proc. VLDB Endowment*, 2012.

[10] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate

relational refinement types for mechanism design and differential privacy. In *Proc. POPL*, 2015.

[11] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proc. USENIX Security*, 2014.

[12] A. Bhowmick, J. Duchi, J. Freudiger, G. Kapoor, and R. Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv:1812.00984 [cs, stat]*, Dec. 2018.

[13] V. Bindschaedler, S. Rane, A. E. Brito, V. Rao, and E. Uzun. Achieving differential privacy in secure multiparty data aggregation protocols on star networks. In *Proc. CODASPY*, Mar. 2017.

[14] 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.

[15] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proc. PODS*, 2005.

[16] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. M. Kiddon, J. Konecny, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards federated learning at scale: System design. In *Proc. SysML*, 2019.

[17] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical Secure Aggregation for Federated Learning on User-Held Data. *arXiv:1611.04482 [cs, stat]*, Nov. 2016.

[18] M. Burkhart, M. Strasser, D. Many, and X. A. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proc. USENIX Security*, 2010.

[19] X. Cao, J. Jia, and N. Z. Gong. Data poisoning attacks to local differential privacy protocols, 2019. arXiv: 1911.02046 [cs.CR].

[20] T.-H. H. Chan, E. Shi, and D. X. Song. Privacy-preserving stream aggregation with fault tolerance. In *Proc. FC*, 2012.

[21] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proc. NSDI*, 2012.

[22] A. Cheu, A. Smith, and J. Ullman. Manipulation attacks in local differential privacy, 2019. arXiv: 1909.09630 [cs.DS].

[23] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proc. NSDI*, Mar. 2017.

[24] G. Danezis, C. Fournet, M. Kohlweiss, and S. Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proc. SEGS*, 2013.

[25] A. A. de Amorim, M. Gaboardi, E. J. Gallego Arias, and J. Hsu. Really natural linear indexed type checking. In *Proc. IFL*, 2014.

[26] A. A. de Amorim, M. Gaboardi, J. Hsu, and S. Katsumata. Probabilistic relational reasoning via metrics. In *Proc. LICS*, 2019.

[27] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *Proc. NIPS*, 2017.

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

[29] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. NOW Publishers, 2014.

[30] U. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *Proc. SODA*, 2019.

[31] U. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proc. CCS*, 2014.

[32] G. Fanti, V. Pihur, and U. Erlingsson. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *arXiv:1503.01214 [cs]*, Mar. 2015.

[33] P. Fox-Penner. A year later, lessons from the blackout. *The New York Times*, Aug. 2004. `https://www.nytimes.com/2004/08/15/opinion/a-year-later-lessons-from-the-blackout.html`.

[34] A. Friedman and A. Schuster. Data mining with differential privacy. In *Proc. SIGKDD*, 2010.

[35] D. Froelicher, P. Egger, J. S. Sousa, J. L. Raisaro, Zhicong Huang, C. Mouchet, B. Ford, and J.-P. Hubaux. UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. In *Proc. PETS*, Oct. 2017.

[36] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *Proc. POPL*, Jan. 2013.

[37] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Privacy-preserving distributed linear regression on high-dimensional data. In *Proc. PETS*, 2017.

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

[39] S. Goryczka and L. Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE Transactions on Dependable and Secure Computing*, 14(5):463–477, 2017.

[40] A. Gupta, K. Ligett, F. McSherry, A. Roth, and K. Talwar. Differentially private combinatorial optimization. In *Proc. SODA*, 2010.

[41] A. Gupta, A. Roth, and J. Ullman. Iterative constructions and private data release. In *Proc. TCC*, 2012.

[42] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. USENIX Security*, Aug. 2011.

[43] S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *Proc. CRYPTO*, 2011.

[44] M. Hardt, B. Recht, and Y. Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *Proc. ICML*, 2016.

[45] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *PVLDB*, 3:1021–1032, 2010.

[46] 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.

[47] HTTP Archive. Report: Page weight. `https://httparchive.org/reports/page-weight`, 2020.

[48] M. Joye and B. Libert. A Scalable Scheme for Privacy-Preserving Aggregation of Time-Series Data. In *Proc. FC*, 2013.

[49] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proc. USENIX Security*, 2012.

[50] KU Leuven COSIC. SCALE-MAMBA. `https://github.com/KULeuven-COSIC/SCALE-MAMBA`.

[51] I. Leontiadis, K. Elkhiyaoui, M. Önen, and R. Molva. PUDA - Privacy and unforgeability for data aggregation. In *Proc. CANS*, 2015.

[52] D. Lie and P. Maniatis. Glimmers: Resolving the privacy/trust quagmire. *Proc. HotOS*, 2017.

[53] LWE estimator tool. `https://bitbucket.org/malb/lwe-estimator/`, commit 3019847.

[54] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Proc. EUROCRYPT*, 2010.

[55] F. McSherry and R. Mahajan. Differentially-private network trace analysis. In *Proc. SIGCOMM*, 2010.

[56] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proc. FOCS*, 2007.

[57] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proc. SIGMOD*, 2009.

[58] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.

[59] J. P. Near, D. Darais, C. Abuah, T. Stevens, P. Gaddamadugu, L. Wang, N. Somani, M. Zhang, N. Sharma, A. Shan, and D. Song. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. In *Proc. OOPSLA*, 2019.

[60] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. *Proc. IEEE Symposium on Security and Privacy*, 2013.

[61] A. Nilsson, P. Nikbakht Bideh, and J. Brorsson. A survey of published attacks on Intel SGX. Available from `https://portal.research.lu.se/portal/files/78016451/sgx_attacks.pdf`, 2020.

[62] Orchard codebase. `https://github.com/edoroth/orchard`.

[63] A. Papadimitriou, A. Narayan, and A. Haeberlen. DStress: Efficient differentially private computations on distributed data. In *Proc. EuroSys*, Apr. 2017.

[64] N. Papernot, M. Abadi, U. Erlingsson, I. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *Proc. ICLR*, 2017.

[65] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[66] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NIPS*. 2019.

[67] Pequin: An end-to-end toolchain for verifiable computation, SNARKs, and probabilistic proofs. `https://github.com/pepper-project/pequi`.

[68] V. Pihur, A. Korolova, F. Liu, S. Sankuratripati, M. Yung, D. Huang, and R. Zeng. Differentially-private "draw and discard" machine learning. *ArXiv*, abs/1807.04369, 2018.

[69] L. Pillaud-Vivien, A. Rudi, and F. Bach. Statistical optimality of stochastic gradient descent on hard learning problems through multiple passes. In *Proc. NeurIPS*, 2018.

[70] W. Qardaji, W. Yang, and N. Li. Understanding hierarchical methods for differentially private histograms. *Proc. VLDB Endow.*, 6(14):1954–1965, Sept. 2013.

[71] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proc. SIGMOD*, 2010.

[72] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proc. ICFP*, 2010.

[73] L. Reyzin, A. Smith, and S. Yakoubov. Turning HATE Into LOVE: Homomorphic Ad Hoc Threshold Encryption for Scalable MPC. `https://eprint.iacr.org/2018/997`, 2018.

[74] E. Rieffel, J. Biehl, W. van Melle, and A. J. Lee. Secured histories: computing group statistics on encrypted data while preserving individual privacy, 2010. arXiv:1012.2152 [cs.CR].

[75] S. Rogers and M. A. Girolami. A first course in machine learning. In *Chapman and Hall / CRC machine learning and pattern recognition series*, 2011.

[76] E. Roth, D. Noble, B. Hemenway Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proc. SOSP*, Oct. 2019.

[77] A. Shamir. How to share a secret. *CACM*, 22(11):612–613, 1979.

[78] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. X. Song. Privacy-preserving aggregation of time-series data. In *Proc. NDSS*, 2011.

[79] J. Tang, A. Korolova, X. Bai, X. Wang, and X. Wang. Privacy loss in Apple's implementation of differential privacy on MacOS 10.12, 2017. arXiv:1709.02753 [cs.CR].

[80] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou. A hybrid approach to privacy-preserving federated learning. In *Proc. 12th ACM Workshop on Artificial Intelligence and Security*.

[81] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In *Proc. CCS*, 2017.

[82] J. Xu, Z. Zhang, X. Xiao, Y. Yang, and G. Yu. Differentially private histogram publication. In *Proc. ICDE*, 2012.

[83] A. Yao. Protocols for secure computations. In *Proc. FOCS*, 1982.

[84] D. Zhang and D. Kifer. LightDP: Towards automating differential privacy proofs. In *Proc. POPL*, 2017.

[85] H. Zhang, E. Roth, A. Haeberlen, B. C. Pierce, and A. Roth. Fuzzi: A three-level logic for differential privacy. In *Proc. ICFP*, Aug. 2019.

[86] J. Zhang, X. Xiao, and X. Xie. Privtree: A differentially private algorithm for hierarchical decompositions. In *Proc. SIGMOD*, 2016.

[87] W. Zhu, P. Kairouz, B. McMahan, H. Sun, and W. Li. Federated heavy hitters discovery with differential privacy, 2019. arXiv:1902.08534.

# A Fuzz

Orchard uses the Fuzz functional programming language designed for differential privacy for constructing queries, and applies additional query transformation steps to produce executable code on its MPC framework.

## A.1 Basic syntax

Fuzz is a higher-order functional programming language, with additional primitives to compute over private datasets known as "bags", and probabilistic commands that takes a private value, adds noise to it and releases the noised value as public value.

The two most fundamental bag operations are `bmap` (bag map), and `bsum` (bag sum). The operation `bmap` takes a function `f` that transforms each individual element in a bag, and produces a new bag whose values are the outputs of `f` applied to each value in the original bag. The operation `bsum` computes the sum of a bag of numbers, after clipping each number in the bag into some range $[-r, r]$. The clipping is necessary to ensure the differential privacy property.

With `bmap` and `bsum`, we can already perform many useful computations over bags. The function `kmeans_iter` counts the number of points in a bag by mapping all points in a bag to the value 1, and sums up this with a clip range of $[-1,1]$.

Another useful bag operation implemented in terms of `bmap` is `bfilter` (bag filter). This operation takes a boolean predicate `f` over values inside a bag, and only keeps elements on which `f` evalutes to `True`. This is implemented through the following Fuzz code by mapping bag values `v` into optional values `Just v` or `Nothing` based on whether `f v` is `True` or `False`:

```
filter_fun f v =
  if f v then Just v else Nothing
bfilter f bag =
  bmap (filter_fun f) bag
```

A final bag operation supplied by Fuzz is `bpartition` (bag partition). The operator `bpartition` takes a known constant that specifies the number of partitions we are creating, a function that maps each bag value into an integer partition index, and the bag to be partitioned. Bag partition then returns a list of sub-bags, and the $i$th sub-bag in the list contains all bag values whose partition index evalutes to $i$. Readers may wonder why `bpartition` requires a known constant parameter for the partition count, since `bpartition` can already infer the partition count from the partition index values. This arrangement is required to keep the number of partitions constant, so that the partition count cannot depend on the bag element values, otherwise bag partition may leak private information about the bag values [57].

In addition to bag operations, Fuzz provides probabilistic commands that act as release mechanisms for private values. In this work, we use the Laplace mechanism `lap` for releasing numerical values, and the exponential mechanism `em` for releasing categorical values.

The function `lap` takes a noise width $w$, and a center $c$. The value `lap c w` represents a Laplace distribution centered at $c$ width $w$, and this distribution can then be sampled from to release a public value from the private center $c$. The exponential mechanism `em` takes a list of private scores with length $n$, and produces a distribution over the integers $[0,n)$. The $i$th scores supplied to `em` indicates "preference" or "quality" of the choice $i$, and the exponential mechanism will select the choice $i$ that approximately maximizes the score, while providing differential privacy protection of the score values.

## A.2 Type System

The term-level syntax and runtime characteristics of Fuzz is an ordinary higher-order functional programming language, with 4 primitive bag operators (`bmap`, `bsum`, `bfilter`, and `bpartition`). However, Fuzz has a unique type system that keeps track of *function sensitivity* and *privacy cost* of programs.

Values in Fuzz are endowed with a distance metric $d(\cdot,\cdot)$. A Fuzz function $f$ with function sensitivity $s$ means if the inputs $x_1$ and $x_2$ satisfy $d(x_1,x_2) \leq 1$, then $d(f(x_1),f(x_2)) \leq s$.

Sensitivity analysis is a key ingredient for determining the privacy cost of a program [29]. By statically determining sensitivities of expressions supplied to release mechanisms (`lap` and `em`), Fuzz's type system can calculate the privacy cost of running the program in many cases. A notable exception is that Fuzz forbids usage of release mechanisms in unbounded loops. This is because the type system cannot statically determine the number of iterations for such loops, and thus cannot compute the total privacy cost for such loops if they contained release mechanisms.