Verifiable Differential Privacy

Arjun Narayan[⋆] Ariel Feldman[‡] Antonis Papadimitriou[⋆] Andreas Haeberlen[⋆]

[⋆]University of Pennsylvania [‡]University of Chicago

Abstract

Working with sensitive data is often a balancing act between privacy and integrity concerns. Consider, for instance, a medical researcher who has analyzed a patient database to judge the effectiveness of a new treatment and would now like to publish her findings. On the one hand, the patients may be concerned that the researcher's results contain *too much* information and accidentally leak some private fact about themselves; on the other hand, the readers of the published study may be concerned that the results contain *too little* information, limiting their ability to detect errors in the calculations or flaws in the methodology.

This paper presents VerDP, a system for private data analysis that provides *both* strong integrity *and* strong differential privacy guarantees. VerDP accepts queries that are written in a special query language, and it processes them only if a) it can certify them as differentially private, and if b) it can prove the integrity of the result in zero knowledge. Our experimental evaluation shows that VerDP can successfully process several different queries from the differential privacy literature, and that the cost of generating and verifying the proofs is practical: for example, a histogram query over a 63,488-entry data set resulted in a 20 kB proof that took 32 EC2 instances less than two hours to generate, and that could be verified on a single machine in about one second.

1. Introduction

When working with private or confidential data, it is often useful to publish some aggregate result without compromising the privacy of the underlying data set. For instance, a cancer researcher might study a set of detailed patient records with genetic profiles to look for correlations between certain genes and certain types of cancer. If such a correlation is found, she might wish to publish her results in a medical journal without violating the privacy of the patients. Similar challenges exist in other areas, e.g., when working with financial data, census data, clickstreams, or network traffic.

In each of these scenarios, there are two separate challenges: privacy and integrity. On the one hand, protecting the *privacy* of the subjects who contributed the data is clearly important, but doing so effectively is highly nontrivial: experience with cases like the AOL search data [1] or the Netflix prize [3] has shown that even sanitized or aggregate data will often reveal more information than intended, especially in the presence of auxiliary information [37]. On the other hand, without the original data it is impossible for others to verify the *integrity* of the results. Mistakes can and do occur (even when the author is a Nobel prize winner [23]), and there have been cases where researchers appear to have fabricated or manipulated data to support their favorite hypothesis [6, 12, 27]; this is why reproducibility is a key principle of the scientific method. Overall, this leaves everyone dissatisfied: the subjects have good reason to be concerned about their privacy, the audience has to blindly trust the analyst that the results are accurate, and the analyst herself is unable to reassure either of them, without violating privacy.

Recent work on *differential privacy* [13, 15] has provided a promising approach to addressing the first challenge. At a high level, differential privacy works by a) preventing the data of any individual subject from having a distinguishable effect on the published result, and by b) adding a small amount of random "noise" from a carefully chosen distribution, e.g., the Laplace distribution. For instance, if the data set contains 47 cancer patients, the analyst might publish that the number is 49 "plus or minus a small amount". (The extra imprecision can be handled in the same way as other forms of imprecision that naturally exist in experimental data, e.g., due to sampling error.) Differential privacy rests on a solid mathematical foundation and can give strong, provable guarantees [15]; also, a number of practical tools for differentially private data analysis have recently been developed, including PINQ [30], Airavat [40], and Fuzz [18, 21].

However, while this approach can reliably protect privacy, it does not help with integrity. Indeed, it makes things worse: *even if* the private data set were made publicly available, it would *still* not be possible to verify the published results, since the analyst can freely choose the noise term. For instance, if there are really 47 cancer patients but the analyst would for some reason prefer if the count were 150, she can simply claim to have drawn the noise term +103 from the Laplace distribution (which has support on all the reals).

This paper proposes a solution to this problem: we present VerDP, a technique for privacy-preserving data analysis that offers *both* strong privacy *and* strong integrity guarantees. Like several existing tools, VerDP provides the analyst with a special query language in which to formulate the

EuroSys'15, April 21–24, 2015, Bordeaux, France.

ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2741948.2741978

computation she would like to perform on the private data set. VerDP then analyzes the query and tries to certify two separate properties: 1) that the query is differentially private, and 2) that there is an efficient way to prove, in zero knowledge, that the query was correctly evaluated on the private data set. The former ensures that the result is safe to publish, while the latter provides a way to verify integrity. To prevent manipulation, VerDP ensures that the analyst cannot control the random noise term that is added to the result, and that this, too, can be verified in zero knowledge.

Our approach builds on our prior work on the Fuzz compiler [18, 39], which uses a linear type system to certify queries as differentially private, as well as on the Pantry system [8] for proof-based verifiable computation. However, a simple combination of Fuzz and Pantry does not solve our problem. First, there are differentially private queries that, if straightforwardly expressed as verifiable programs, would leak private information through the programs' structures (e.g. those that employ data-dependent recursion). We address this problem by creating a verifiable subset of Fuzz, called VFuzz, that prevents these leaks without substantially compromising expressiveness. Second, because differential privacy is typically used with large data sets, a naïvely constructed verifiable program would be enormous, and constructing proofs would take far too long. To overcome this challenge, we break queries down into smaller components that can be proved more quickly and that can take advantage of parallelism and batch processing. This allows us to amortize the high setup costs of verifiable computation.

We built a prototype implementation of VerDP, and evaluated it by running several differentially private queries that have been discussed in the literature, from simple counting queries to histograms and k-means clustering. We found that a histogram query on a private data set with 63,488 entries resulted in a 20 kB proof that took 32 Amazon EC2 GPU instances less than two hours to generate (an investment of approximately $128, at current prices) and that could be verified on one machine in about one second. In general, our results show that a research study's results can be verified quite efficiently even without access to fast networks or powerful machines. Moreover, the cost of constructing a proof, while high, is not unreasonable, given that it can be done offline and that it only needs to be done once for each research study. Proof generation time can be further improved by adding machines, as VerDP is highly parallelizable.

VerDP does not protect against malicious analysts who intentionally leak private data. But it makes experiments based on private data verifiable and repeatable for the first time — a key requirement for all experimental science. Our main contributions are as follows:

- VFuzz, a query language for computations with certifiable privacy and verifiable integrity (Section 4);
- The design of VerDP, a system for verifiable, differentially private data analysis (Section 5); and
- A prototype implementation of VerDP and its experimental evaluation (Sections 6 and 7).

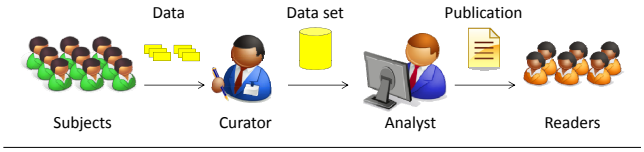

Figure 1. The scenario we address in this paper.

2. Overview

Figure 1 illustrates the scenario we address in this paper. A group of *subjects* makes some sensitive data, such as information about their income or their health, available to a *curator*. The curator then grants access to the resulting data set *db* to a (carefully vetted) group of *analysts*, who study the data and then publish their findings to a group of *readers*. This separation between curator and analyst reflects a common practice today in large-scale studies using sensitive data, where the raw data is often hosted by a single organization, such as the census bureau, the IPUMS data sets [28], the iDASH biomedical data repository [26], or the ICPSR data deposit archive [25]. Each analyst's publication contains the results of at least one *query q* that has been evaluated over the private data set *db*; typical examples of queries are aggregate statistics, such as the average income of a certain group, or the number of subjects that had a certain genetic condition.

We focus on two specific challenges. First, the subjects may be concerned about their *privacy*: the published result $q(db)$ might accidentally contain "too much" information, so that readers can recover some or all of their sensitive data from it. This concern could make it more difficult for the curator to recruit subjects, and could thus restrict the data sets that are available to potential analysts. Second, some readers may be concerned about *integrity*: a dishonest analyst could publish a fake or biased $q(db)$, and a careless analyst could have made mistakes in her calculations that would invalidate the results. Since the data set *db* is sensitive and the readers do not have access to it, the analyst has no way to alleviate such concerns.

2.1 Goals

In designing a system for this scenario, we focus on enforcing two properties:

- **Certifiable privacy:** It should be possible to formally prove, for a large class of queries q, that the result $q(db)$ does not leak too much information about any individual subject.
- **Verifiable integrity:** It should be possible to verify that a published result $q(db)$ is consistent with the private data set *db*, *without* leaking any data that is not already contained in $q(db)$.

These properties help all four parties: The curator could promise the subjects that all published results will be certified as private, which could alleviate their privacy concerns and help the curator recruit subjects more easily. The analyst could be sure that her published results are not deanonymized later and thus expose her to embarrassment and potential liability. Finally, interested readers could verify the published results to gain confidence that they have been properly derived from the private data — something that is not currently possible for results based on data that readers are not allowed to see.

2.2 Threat model

The primary threat in our scenario comes from the analyst. We consider two types of threats: a *blundering analyst* accidentally publishes results that reveal too much about the data of some subjects, while a *dishonest analyst* publishes results that are not correctly derived from the private data set. Both types of threats have been repeatedly observed in practice [1, 6, 12, 27, 37]; we note that dishonest analysts would be much harder to expose if differentially private studies become more common, since the addition of 'random' noise would give them plausible deniability. In this work, we do *not* consider analysts that leak the raw data: information flow control is an orthogonal problem to ours, which focuses on declassified (and published) data.

We rely on the separation of duties between the curator (who collects the data) and the analyst (who studies it) to prevent a dishonest analyst from maliciously choosing subjects in order to produce a desired study result. This seems reasonable because real-world data collectors, like the Census Bureau, serve many analysts and are not likely to have a vested interest in the outcome of any one analyst's study. Besides, the data set may be collected, and publicly committed to, long before a given analyst formulates her query. This is common today in many social science and medical research projects.

We assume that the set of readers may contain, now or in the future, some curious readers who will try to recover the private data of certain subjects from the published results $q(db)$. The curious readers could use the latest deanonymization techniques and have access to some auxiliary information (such as the private data of certain other subjects). However, we assume that readers are computationally bounded and cannot break the cryptographic primitives on which VerDP is based.

A dishonest analyst might attempt to register many variants of the same query with the curator, with the hope of obtaining results with different noise terms, and then publish only the most favorable result. However, it should not be difficult for the curator to detect and block such attacks.

2.3 Strawman solutions

Intuitively, it may seem that there are several simple solutions that would apply to our scenario. However, as illustrated by several high-profile privacy breaches [1, 3], intuition is often not a good guide when it comes to privacy; simple solutions tend to create subtle data leaks that can be exploited once the data is published [37]. To illustrate this point, and to motivate the need for solid formal foundations, we discuss a few strawman solutions below.

```
function kMeansCluster (db: data) {
  centers := chooseInitialCenters();
  repeat
    centers := noise(update(db, centers));
  until (centers.converged);
  return centers;
}
```

Figure 2. A very simple program for which a naïvely constructed ZKP circuit would leak confidential information.

Trusted party: One approach would be to simply have the curator run the analyst's query herself. This is undesirable for at least two reasons. First, such a system places all the trust in the curator, whereas with VerDP, fraud requires collusion between the curator and the analyst. Second, such a design would not allow readers to detect non-malicious errors in the computation, as they can with VerDP.

ZKP by itself: Another approach would be to use an existing tool, such as ZQL [16], that can compile computations on private data into zero-knowledge proofs (ZKPs). Although such systems would prevent readers from learning anything but a query's results, they do not support differential privacy, and so they would not provide any meaningful limit on what readers could infer from the results themselves. For example, suppose that the private data db consists of the salary s_i of each subject i, and that the analyst publishes the average salary $q(db) := (\sum_i s_i)/|db|$. If an adversary wants to recover the salary s_j of some subject j and already knows the salaries of all the other subjects, he can simply compute $s_j := |db| \cdot q(db) - \sum_{i \neq j} s_i$. In practice, these attacks can be a lot more subtle; see, e.g., [37].

Naïve combination of DP and ZKP: A third approach would be to modify an existing differentially private data analysis tool, such as PINQ [30] or Airavat [40], to dynamically output a circuit instead of executing a query directly, and to then feed that circuit to a verifiable computation system like Pinocchio [38]. Performance would obviously be a challenge, since a naïve translation could produce a large, unwieldy circuit (see Section 5.4); however, a much bigger problem with this approach is that *private data can leak through the structure of the circuit*, which Pinocchio and similar systems assume to be public. Figure 2 shows a sketch of a very simple program that could be written, e.g., in PINQ, to iteratively compute a model that fits the private data. But since the number of iterations is data-dependent, no finite circuit can execute this program for *all* possible inputs – and if the circuit is built for the actual number of iterations that the program performs on the private data, an

attacker could learn this number by inspecting the circuit, and then use it to make inferences about the data. To reliably and provably prevent such indirect information leaks, it is necessary to carefully co-design the analysis tool and the corresponding ZKP, as we have done in VerDP.

3. Background

3.1 Differential Privacy

Differential privacy [15] is one of the strongest privacy guarantees available. In contrast to other solutions such as k-anonymity [44], differential privacy offers provable guarantees on the amount of information that is leaked, regardless of the auxiliary knowledge an adversary may have.

Differential privacy is a property of *randomized functions* that take a database as input, and return an aggregate output. Informally, a function is differentially private if changing any single row in the input database results in *almost* no change in the output. If we view each row as consisting of the data of a single individual, this means that any single individual has a statistically negligible effect on the output. This guarantee is quantified in the form of a parameter ϵ, which corresponds to the amount that the output can vary based on changes to a single row. Formally, for any two databases d_1 and d_2 that differ only in a single row, we say that f is *ϵ-differentially private* if, for any set of outputs R,

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

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

The standard method for achieving differential privacy for numeric queries is the *Laplace mechanism* [15], 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/ϵ, which results in $\epsilon-$differential privacy. (Most empirical studies already deal with some degree of "noise", e.g., due to sampling error, and the extra noise from the Laplace mechanism can be handled in the same way, e.g., with hypothesis testing [24].) Calculating the sensitivity of a given program is non-trivial, and is often the hardest part of building a differentially private system.

Fortunately, differential privacy has two additional properties that make it tractable to reason about. First, it is composable with arbitrary post-processing, with no further loss of privacy. As a result, once we have built a core set of primitives whose output is provably differentially private, we can freely post-process the results of those primitives, as long as the post-processing does not touch the private database. Second, if we evaluate two functions f_1 and f_2 that are ϵ_1- and ϵ_2-differentially private, respectively, publishing both results is at most $(\epsilon_1 + \epsilon_2)$-differentially private. In the differential privacy literature, 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* $\epsilon_{\max}$ 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 $\epsilon_{\max}$, see, e.g., [24].

As originally proposed [15], differential privacy is an information-theoretic guarantee that holds even if the adversary has limitless computational power. However, from an implementation perspective, this would rule out the use of most cryptographic primitives, which typically (and reasonably) assume that the adversary cannot perform certain tasks, such as factoring large numbers. Mironov et al. [33] suggested a relaxation called *computational differential privacy* that assumes a computationally bounded adversary, and this is the definition we adopt in this paper.

3.2 Proof-based verifiable computation

VerDP builds on recent systems for *proof-based verifiable computation* [8, 38]. These systems make it possible for a *prover* $\mathcal{P}$ to run a program Ψ on inputs x and to then provide a *verifier* $\mathcal{V}$ with not only the results $\Psi(x)$, but also with a proof π that demonstrates, with high probability, that the program was executed correctly. (In our scenario, the analyst would act as the prover, the program would be the query, and any interested reader could become a verifier.) This correctness guarantee only depends on cryptographic hardness assumptions, and not on any assumptions about the prover's hardware or software, which could be arbitrarily faulty or malicious. Furthermore, these systems are general because they typically offer compilers that transform arbitrary programs written in a high-level language, such as a subset of C, into a representation amenable to generating proofs. This representation, known as *constraints*, is a system of equations that is equivalent to $\Psi(x)$ in that proving knowledge of a satisfying assignment to the equations' variables is tantamount to proving correct execution of the program.

VerDP uses the Pantry [8] system for verifiable computation, configured with the Pinocchio [38] proof protocol. Together, Pantry and Pinocchio have two other crucial properties. First, unlike other systems that require the verifier to observe the program's entire input, Pantry allows the verifiable computation to operate on a database stored only with the prover, as long as the verifier has a *commitment* to the database's contents. (In Pantry, a commitment to a value v is $\textsc{Comm}(v) := \text{HMAC}_r(v)$, i.e., a hash-based message authentication code of v with a random key r. $\textsc{Comm}(v)$ *binds* the prover to v – he can later open the commitment by revealing v and r, but he can no longer change v – and it also *hides* the value v until the commitment is opened.) Second, Pinocchio enables the proof of computation to be non-interactive and zero-knowledge. As a result, once the proof is generated, it can later be checked by any verifier, and the verifier learns nothing about the program's execution or the database it operated on, other than what program's output implies. If the

computation described by the program is differentially private — as is true of all well-typed VFuzz programs [39] — and if the program itself is compiled without looking at the private data, then neither the output nor the structure of the proof can leak any private data.

The full details of Pinocchio's proof protocol are beyond the scope of this paper, but there is one aspect that is relevant here. In Pinocchio's proof protocol, some party other than the prover (e.g., the verifier) first generates a public *evaluation key (EK)* to describe the computation, as well as a small *verification key (VK)*. The prover then evaluates the computation on the input x and uses the EK to produce the proof π; after that, anyone can then use the VK to check the proof.

3.3 Computing over a database

As we mention above, Pantry allows Ψ to compute over a database that is not included in the inputs x and outputs $\Psi(x)$, which, in VerDP's case, comes from the curator. Pantry makes this possible by allowing the prover $\mathcal{P}$ to read and write arbitrarily-sized data blocks from a block store. To prevent $\mathcal{P}$ from lying about blocks' contents, blocks are named by the collision resistant hashes of their contents, and the compiler transforms each read or write into a series of constraints corresponding to computing the block's hash and comparing it to the expected value (i.e. the block's name). In this way, a computationally-bound $\mathcal{P}$ can only satisfy the constraints C if it uses the correct data from the block store.

Ψ can compute new block names during its execution and retrieved blocks can contain the names of other blocks, but the name of at least one block must be known to the verifier $\mathcal{V}$. One possibility is to include the database's root hash in the input x that $\mathcal{V}$ observes. But, in VerDP, readers cannot be allowed to see the root hash because it could leak information about the database's contents. Instead, x only contains a cryptographic commitment to the database's root hash supplied by the curator [8, §6]. The commitment binds the analyst to the database's contents while hiding the root hash from readers. Furthermore, the compiler inserts constraints into C that are only satisfiable if the analyst correctly opens the commitment to the database's root hash.

3.4 Performance of verifiable computation

Despite recent improvements, verifiable computation systems impose significant overhead on $\mathcal{P}$, often by a factor of up to 10^5 [8]. For verification, however, each new program has high setup costs, but every subsequent execution of that program (on different inputs) can be verified cheaply. Creating an EK and a VK takes time linear in the number of steps in the program — often tens of minutes for the curator — but then readers can verify π in seconds. Not surprisingly, MapReduce-style applications, where a small program is executed many times over chunks of a larger data set, are ideally suited to these limitations, and VerDP exploits this fact (see Section 5.2). Note that commitments are relatively costly in Pantry, and so VerDP uses them sparingly.

4. The VFuzz language

VerDP's query language is based on the Fuzz language for differentially private data analysis [18, 21, 39] because: 1) it is sufficiently expressive to implement a number of practical differentially-private queries, and 2) it uses static analysis to certify privacy properties without running the query or accessing the data.

4.1 The Fuzz query language

We begin by briefly reviewing the key features of the Fuzz query language. Fuzz is a higher-order, dependently typed functional language; queries are written as functions that take the data set as an argument and return the value that the analyst is interested in. Fuzz's type system for inferring the sensitivity of functions is based on linear types [39]. It distinguishes between ordinary functions $\mathtt{f} : \tau \to \sigma$ that can map arbitrary elements of type τ to arbitrary elements of type σ, and functions $\mathtt{f} : \tau \multimap_k \sigma$ that have an upper bound k on their sensitivity – in other words, a change in the argument can be amplified by no more than k in the result. The sensitivity is used to calculate how much random noise must be added to `f`'s result to make it differentially private.

Fuzz offers four primitive functions that can operate directly on data sets: `map`, `split`, `count`, and `sum`. `map` : $\tau\ bag \to (\tau \to \sigma) \multimap \sigma\ bag$[1] applies a function of type $\tau \to \sigma$ to each element of type τ, whereas `split` : $\tau\ bag \to (\tau \to bool) \multimap \tau\ bag$ extracts all elements from a data set d that match a certain predicate of type $\tau \to bool$. With the restriction that $\tau\ bag = \sigma\ bag = db$ both functions take a database and a predicate function of appropriate type, and return another data set as the result. `count` $d : db \to \mathbb{R}$ returns the number of elements in data set d, and `sum` $d : db \to \mathbb{R}$ sums up the elements in data set d; both return a number.

Fuzz also contains a *probability monad* $\bigcirc$ that is applied to operations that have direct access to private data. The only way to return data from inside the monad is to invoke a primitive called `sample` that adds noise from a Laplace distribution based on the sensitivity of the current computation. Reed et al. [39] has shown that any Fuzz program with the type $db \to \bigcirc\tau$ is provably differentially private; intuitively, the reason is that the type system prevents programs from returning values unless the correct amount of noise has been added to them.

Figure 3 shows a "hello world" example written in Fuzz that returns the (noised) number of persons in a data set whose age is above 40. `over_40` is a function with type $row \to bool$. The `split` primitive maps this function over each individual row, and counts the rows where the result of `over_40` was `true`. This result is noised with `sample`, and returned as the final output.

[1] Fuzz uses the type *bag* to refer to multisets. A $\tau\ bag$ is a multiset consisting of τ elements.

```
function over_40(r : row) : bool {
  r.age > 40
}

function main (d :[1] db) : fuzzy num {
  let peopleOver40 = split over_40 d in
    return sample fuzz count peopleOver40
}
```

Figure 3. A simple program written in Fuzz that counts the individuals over 40 in a database of individuals' ages.

4.2 From Fuzz to VFuzz

Fuzz already meets one of our requirements for VerDP: it can statically certify queries as differentially private, without looking at the data. However, Fuzz programs cannot directly be translated to circuits, for two different reasons: 1) the size of the data set that results from a `split` can depend on the data, and 2) `map` and `split` allow arbitrary functions and predicates, including potentially unbounded, data-dependent recursion. Thus, if the execution of a Fuzz program were naïvely mapped to a circuit, the size and structure of that circuit could reveal facts about the private input data.

Our solution to this problem consists of two parts. To address the first problem, all internal variables of type *db* have a fixed size equal to that of the input database, and VFuzz uses a special "empty" value that is compatible with `map` and `split` but is ignored for the purposes of `count` and `sum`. To address the second problem, we require `map` functions and `split` predicates to be written in a restricted subset of C that only allows loops that can be fully unrolled at compile time and disallows unbounded recursion.

We refer to the modified language as Verifiable Fuzz or *VFuzz*. Clearly, not all Fuzz programs can be converted into VFuzz programs, but all practical Fuzz programs we are aware of can be converted easily (for more details, see Section 7.1). VFuzz is slightly less convenient than Fuzz because the programmer must switch between two different syntaxes, but `map` and `split` functions are usually small and simple. They could be automatically translated to C in most cases, using standard techniques for compiling functional programs efficiently, but this is beyond the scope of the present paper.

We note that there is an interesting connection between the above problem and the timing side channels in Fuzz programs that we identified (and fixed) in [21]: the data dependencies that cause the circuit size to change are the same that also affect query execution time, and vice versa. [21] takes a different approach to the second problem: it introduces timeouts on `map` functions and `split` predicates, and if the timeout is exceeded, the function or predicate aborts and returns a default value. In principle, we could use this approach in VerDP as well; for instance, we could introduce a cap on the number of constraints per `map` or `split`, unroll the function or predicate until that limit is reached, and return a default value if termination does not occur before then. Although this approach would enable VFuzz to be more expressive, it would yield a highly inefficient set of constraints.

4.3 Are VFuzz programs safe?

With these changes, all VFuzz programs can be translated to circuits safely, without leaking private data. To see why, consider that all VFuzz programs could be partitioned into two parts: one that operates on values of type *db* (and thus on un-noised private data) and another that does not. We refer to the former as the "red" part and to the latter as the "green" part. Valid programs begin in red and end in green, but they can alternate between the two colors in between – e.g., when a program first computes a statistic over the database (red), samples it (green), and then computes another statistic, based on the sampled output from the earlier statistic (red). The boundary from red to green is formed by `count` and `sum`, which are the only two functions that take a *db* as an argument and return something other than *db* [21].

Crucially, all the red parts can be translated statically because 1) the only values they can contain outside of `map` and `split` are of type *db*, which has a fixed size; 2) the functions and predicates in VFuzz's `map` and `split` are written directly in restricted C; and 3) the four data set primitives (`map`, `split`, `count`, and `sum`) can be translated statically because they simply iterate over the entire data set. We emphasize that, because circuits are generated directly from VFuzz programs without looking at the private database, the resulting circuits cannot depend on, and thus cannot leak, the private data in any way.

What about the green parts? These are allowed to contain data-dependent recursion (and generally all other features of the original Fuzz), but the only way that data can pass from red to green is by sampling, which adds the requisite amount of noise and is therefore safe from a privacy perspective. (Unlimited sampling is prevented by the finite privacy budget.) Indeed, since the green parts of a VFuzz program can only look at sampled data, and the differential privacy guarantees remain even with arbitrary post-processing [15], *there is no need to translate the green parts to circuits at all* – the analyst can simply publish the sampled outputs of the red parts, and the readers can re-execute the green parts, e.g., using the standard Fuzz interpreter, or any other programming language runtime, incurring no additional cryptographic overhead. A proof of VFuzz's safety is omitted here due to space constraints, but is available in the extended version [35].

5. The VerDP system

This section describes VerDP's design and how it realizes verifiable, privacy-preserving data analysis for queries written in VFuzz.

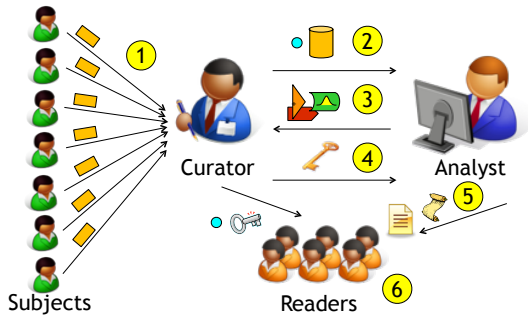

Figure 4. Workflow in VerDP. The numbers refer to the steps in Section 5.1.

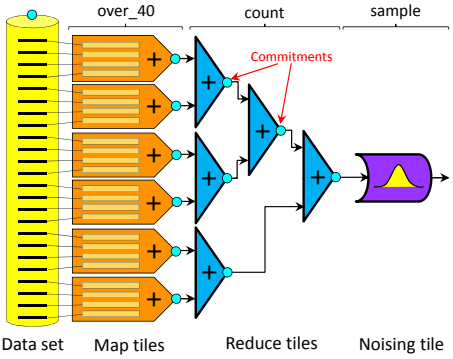

Figure 5. The MapReduce-like structure of VFuzz programs. The small circles represent commitments, and the labels at the top show the parts of the `over40` program from Figure 3 that correspond to each phase.

5.1 VerDP's workflow

Figure 4 illustrates VerDP's workflow, which consists of the following steps:

1. The database curator collects the private data of each subject into a data set *db*, and then publishes a commitment Comm(db) to it (Section 5.3). The curator also creates and maintains a privacy budget for the data set.
2. An analyst requests the data set from the curator and is vetted. If her request is granted, the analyst studies the data, formulates a hypothesis, and decides on query q for which she wants to publish the results.
3. The analyst submits q to the curator,[2] who compiles and typechecks it with the VFuzz compiler to ensure that it is differentially private. If q fails to typecheck or exceeds *db*'s privacy budget, the analyst must reformulate q.
4. If q is approved, the compiler converts all of the "red" portions of q into a series of verifiable programs $(\Psi_1, \ldots, \Psi_m)$ (see Section 5.2). For each Ψ_i, the curator generates an evaluation key EK_i and a verification key VK_i, and gives the EKs to the analyst while making the VKs public. Finally, the curator generates a random seed r, gives it to the analyst, and publishes a commitment to it Comm(r).
5. The analyst runs q by executing the verifiable programs and publishes the result $q(db)$. She adds noise to $q(db)$ by sampling from a Laplace distribution using a pseudorandom generator seeded with r. In addition, every time she runs Ψ_i — and she often runs a given verifiable program multiple times (see below) — she uses the corresponding EK_i to produce a proof π that the program was executed correctly (Section 5.4). She then publishes the proofs π_i. Finally, for every value v passed from one verifiable program to another, she publishes a commitment Comm(v).
6. Readers who wish to verify the analyst's results obtain q and begin running its "green" portions. Every time they reach a "red" portion, they obtain the corresponding proofs, commitments, and verification keys and check that that portion of q was executed correctly (Section 5.5). If they successfully verify all of the "red" portions of q and obtain the same result as the analyst after running the "green" portions, they accept the results.

Next, we explain each of these steps in more detail.

5.2 The structure of VFuzz programs

VerDP's design is guided by both the properties of the VFuzz language (Section 4) and the performance characteristics of verifiable computation (Section 3.4). On the one hand, we observe that the "red" portions of every VFuzz query have a similar MapReduce-like structure: first, there is a map phase where some combination of `map` functions and `split` predicates are evaluated on each row of the data set independently, and second, there is a reduce phase where some combination of `count` and `sum` operators aggregate the per-row results. Finally, there is a phase that adds random noise to this aggregate value. Returning to the example in Figure 3, the map phase corresponds to the `split` operator with the `over40` predicate, the reduce phase corresponds to the `count` operator, and the noise phase corresponds to the `sample` operator. On the other hand, verifiable computation is most efficient, not when there is a large monolithic program, but in a MapReduce setting where a small program is executed many times over chunks of a larger data set, thereby amortizing the high setup costs.

These observations led us to the design shown in Figure 5. The VFuzz compiler converts each phase of every "red" section into its own independent verifiable program, written in a restricted subset of C. These three verifiable programs are compiled separately, and the curator generates a separate EK and VK for each one. To amortize each program's setup costs, we exploit the fact that both the map and reduce phases are embarrassingly parallel: their inputs can be partitioned and processed independently and concurrently. Thus, the

[2] VerDP actually enables a further separation of duties: because they can be created solely from q, the EKs could be generated by a party other than the curator, which never has to see *db* at all.

map and reduce programs only operate on chunks of their phase's input, and the analyst processes the entire input by running multiple map and reduce instances, potentially in parallel across multiple cores or machines.[3] We refer to these instances as *tiles* because VerDP must run enough tiles to cover the entire data flow. For each tile that she executes, the analyst produces a proof π that readers must verify. Fortunately, these proofs are small and cheap to verify.

If a VFuzz query is broken down into separate verifiable programs, how does the analyst prove to readers that she has correctly used the output v of one phase as the input to the next without revealing un-noised intermediate values? The answer is that, for each tile in each phase, she publishes a commitment Comm(v) to the tile's output, which is given as input to a tile in the next phase (blue circles in Figure 5). The analyst can only satisfy the constraints of each phase's verifiable program if she correctly commits and decommits to the values passed between phases. Readers can later use the Comm(v)s along with the proofs (πs) for each instance in each phase to verify the "red" portions of the query.

5.3 Committing to the data set

Recall our verifiable integrity goal: the readers should be able to verify that some published result r was produced by evaluating a known query q over a private database db – *without* having access to the database itself! To make this possible, the curator publishes a commitment to the data set Comm(db), and the constraints of every mapper program are chosen so that they can only be satisfied if the analyst actually uses the data set corresponding to Comm(db) when executing the query (see Section 3.3). In other words, the analyst must prove that she knows a database db that a) is consistent with Comm(db), and b) produces $r = q(db)$.

In principle, the curator could commit to the flat hash of the data set. But, in that case, each mapper would have to load the entire data set in order to check its hash. As a result, each mapper program would require $\Omega(|db|)$ constraints, and the mappers would be too costly for the analyst to execute for all but the smallest data sets. For this reason, the curator organizes the data set as a hash tree where each tree node is a verifiable block (see Section 3.3), and each leaf node contains the number of rows that an individual map tile can process; we refer to the latter as a *leaf group*. The curator can then commit to the root hash of this tree. In this way, each map tile only needs $O(\log|db|)$ constraints.

Because all loop bounds and array sizes must be determined statically, the number of rows in each leaf must be fixed at compile-time. The number of rows per leaf must be chosen carefully, as it affects the runtime of the map tiles. Having too few per leaf is inefficient because even opening the commitment to the db's root hash incurs a high cost that is best amortized across many rows. But having too many rows per leaf results in a map tile that takes to long to execute and exhausts the available memory. As each map tile processes at least one leaf, the number of leaves represents an upper bound on the available parallelism.

Beside committing to the data set, the curator also commits to a random seed r that the analyst uses to generate noise (see Section 5.4). This seed *must* be private because if it were known to the readers, then they could recompute the noise term, subtract it from the published result, and obtain the precise, un-noised result of the query. The seed could be chosen by the curator, or the research subjects could generate it collaboratively.

5.4 Phases of VerDP computations

Recall from Section 4.3 that VFuzz programs consist of red and green parts. Before VerDP can generate tiles for a query, it must first identify all the red parts of the query; this can be done with a simple static analysis that traces the flow of the input data set db to the aggregation operators (`count` and `sum`) that mark the transition to green.

The map phase: Every mapper program has a similar structure. It has at least two inputs: Comm(db) and an integer that identifies which leaf of the data set tree it should process. It then proceeds as follows. First, it verifiably opens Comm(db) and then retrieves its leaf of the data set by fetching $O(\log|db|)$ verifiable blocks. Second, it performs a sequence of `map` and `split` operations on each row in its leaf independently. Third, it aggregates the results of these operations across the rows in its leaf using either the `sum` or the `count` operator. Finally, it outputs a commitment to this local aggregate value that is passed along to the reduce phase.

The reduce phase: Ideally, the reduce phase could just consist of a single instance of a single program that verifiably decommitted to all of the map tiles' outputs, computed their sum, and then committed to the result. Unfortunately, the high cost of verifiably committing and decommitting means that each reduce tile can only handle a handful of input commitments. Consequently, VerDP must build a "tree" of reduce tiles that computes the global sum in several rounds, as illustrated in Figure 5. Thus, if k is the number of map tiles, then the reduce phase consists of $O(\log k)$ rounds of reduce tiles. However, within each round except the last one, multiple reduce tiles can run in parallel. In our experiments (see Section 7), we use reducers that take only two commitments as input in order to maximize the available parallelism.

Notably, all VFuzz queries use the same programs for their reduce and noise phases. Thus, the curator only has to generate an EK and VK for these programs once for all analysts, substantially reducing the work required to approve an analyst's query.

The noise phase: The noise phase adds random noise drawn from a Laplace distribution to the aggregate results of the previous phases. The single noise tile has four inputs: 1) the sensitivity s of the computation whose result is being

[3] Pantry's MapReduce implementation exploits similar parallelism [8, §4], but VerDP's use case is even more embarrassingly parallel. Whereas in Pantry, each mapper has to produce an output for every reducer, in VerDP each mapper instance only needs to send input to a single reducer instance.

noised, 2) the privacy cost ϵ of that particular computation, 3) the curator's commitment Comm(db) to the original data set, and 4) Comm(r), the curator's commitment to a random 32-bit seed r. The noise tile works by first converting the seed to a 64-bit fixed-point number $\bar{r}$ between 0 and 1, and then computing $\frac{s}{\epsilon} \cdot \ln(\bar{r})$, using one extra bit of randomness to determine the sign. If the input is in fact chosen uniformly at random, then this process will result in a sample from the $Lap(s/\epsilon)$ distribution. Since our verifiable computation model does not support logarithms natively, our implementation approximates $\ln(\bar{r})$ using 16 iterations of Newton's method. We also use fixed-point (as opposed to floating-point) arithmetic to avoid an attack due to Mironov [32], which is based on the fact that the distance between two adjacent floating-point values varies with the exponent. To provide extra protection against similar attacks, our noise generator could be replaced with one that has been formally proven correct, e.g., an extension of the generator from [22] or the generation circuit from [14].

5.5 Proof verification

An interested reader who wants to verify the analyst's computation needs four ingredients: 1) the curator's commitment to the data set the analyst has used; 2) the exact VFuzz code of the queries; 3) the VK and the query index that the curator has generated for the analyst; and 4) the analyst's proof.

The reader begins by using VerDP to type-check each of the queries; if this check fails, the analyst has failed to respect differential privacy. If the type check succeeds, VerDP will (deterministically) compile the query into the same set of tiles that the analyst has used to construct the proof. The reader now verifies the proof; if this fails, the analyst has published the wrong result (or the wrong query).

If the last check succeeds, the reader has established that the red parts of the published query were correctly evaluated on the data set that the curator collected, using a noise term that the analyst was not able to control. As a final step, the reader plugs the (known but noised) output values of each red part into the green part of the VFuzz program and runs it through an interpreter. If this final check succeeds, the reader can be satisfied that the analyst has indeed evaluated the query correctly, and since the proof was in zero knowledge, he has not learned anything beyond the already published values. Thus, VerDP achieves the goals of certifiable privacy and verifiable integrity we have formulated in Section 2.1.

5.6 Limitations

VerDP's integrity check is limited to verifying whether a query q, given as a specific VFuzz program, will produce a certain result when evaluated over the data set that the curator has committed to. This does *not* mean that q actually does what the analyst claims it will do – a blundering analyst may have made a mistake while implementing the query, and a dishonest analyst may intentionally formulate an obfuscated query that appears to do one thing but actually does another. In principle, the readers can detect this because the code of the query is available to them, but detecting such problems is not easy and may require some expertise.

Not all differentially private queries can be expressed in VFuzz, for at least two reasons. First, there are some useful primitives, such as the GroupBy in PINQ [30], that are not supported by the original Fuzz, and thus are missing in VFuzz as well. This is not a fundamental limitation: Fuzz can be extended with new primitives, and these primitives can be carried over to VFuzz, as long as the structure of the corresponding proofs does not depend on the private data. Second, it is known [18] that some queries cannot be automatically certified as differentially private by Fuzz or VFuzz because the proof relies on a complex mathematical truth that the type system fails to infer. This is because Fuzz, as a system for non-experts, is designed to be automated as much as possible. [18] shows that some of these limitations can be removed by extending the type system; also, analysts with more expertise in differential privacy could use a system like CertiPriv [2] to complete the proof manually. However, these approaches are beyond the scope of this paper.

6. Implementation

VerDP builds upon two core programs—the VFuzz compiler and typechecker, for certifying programs as differentially private, and the Pantry verifiable computation system, for compiling to a circuit, generating zero knowledge proofs of correct circuit execution, and verifying the results of the generated proofs. Verifiable programs are written in a restricted subset of C without recursion or dynamic loops.

Our VFuzz compiler is based on our Fuzz compiler from [21]; we modified the latter to emit functions in restricted C, and we wrote some additional libraries in C to support the privileged Fuzz operators `map` and `split`, as well as for generating Laplace noise. We also modified the Fuzz compiler to accept VFuzz programs as input, as detailed in Section 4.

A single VFuzz query results in multiple verifiable programs, depending on the number of `sample` calls. For a given `sample`, the VFuzz compiler outputs a map tile program, which is run in parallel on each leaf group. The reduce tile and noising tile programs are fixed and identical across all queries. A single `sample` requires a tree of reducers, but each individual component of the reducer is the same. The tree is run until there is a single output, which is the actual un-noised result of the `sample` call in VFuzz. The final step is to input this value to the Laplace noising program, which outputs the final noised result.

For verification, each individual tile needs to be checked separately. However, since the outputs are not differentially private, only *commitments* to the outputs are sent. Only the final output from the noising tile is differentially private and thus sent in the clear.

Query	Type	LoC mod.	Samples	From
over-40	Counting	2 / 45	1	[15]
weblog	Histogram	2 / 45	2	[15]
census	Aggregation	9 / 50	4	[9]
kmeans	Clustering	46 / 148	6	[7]

Table 1. Queries we used for our experiments (based on [21]), lines of code modified, and the inspirations.

The VFuzz compiler ignores post-processing instructions. Fuzz allows for arbitrary computation outside the probability monad for pretty-printing, etc., but these are not necessary to protect privacy. Since it would be expensive (and unnecessary) to compute these post-processing steps as verifiable programs, we discard them during proof generation and simply re-execute them during verification.

7. Evaluation

Next, we report results from an experimental evaluation of our VerDP prototype. Our experiments are designed to answer three key questions: 1) Can VerDP support a variety of different queries?, 2) Are the costs low enough to be practical?, and 3) How well does VerDP scale to larger data sets?

7.1 Queries

We used four different queries for our experiments. The first three are the queries that were used to evaluated Fuzz [21]: **weblog** computes a histogram over a web server log that shows the number of requests from specific subnets; **census** returns the income differential between men and women on a census data set; and **kmeans** clusters a set of points and returns the three cluster centers. Each of these queries is motivated by a different paper from the privacy literature [7, 9, 15], and each represents a different type of computation (histogram, aggregation, and clustering). We also included our running example from Figure 3 as an additional fourth query; we will refer to it here **over-40** because it computes the number of subjects that are over 40 years old. Table 1 shows some statistics about our four queries.

Since VerDP's query language differs from that of Fuzz, we had to modify each query to work with VerDP. Specifically, we re-implemented the mapping function of each `map` and the predicate of each `split` in our safe subset of C. As Table 1 shows, these modifications were minor and only affected a very few lines of code, and mostly involved syntax changes in the mapping functions from functional to imperative. Importantly, the changes from Fuzz to VFuzz do not reduce the expressiveness of the queries nor degrade their utilities. We also inspected all the other example programs that come with Fuzz, and found that each could have been adapted for VerDP with small modifications to the code; none used constructs (such as unbounded recursion) that VerDP does not support. This suggests that, in practice, the applicability of VerDP's query language is comparable to Fuzz.

7.2 Experimental setup

Since Pantry can take advantage of GPU acceleration to speed up its cryptographic operations, and since VerDP can use multiple machines to run map and reduce tiles in parallel, we use 32 `cg1.4xlarge` instances on Amazon EC2 for our experiments. This instance type has a 64-bit Intel Xeon x5570 CPU with 16 virtual cores, 22.5 GB of memory, and a 10 Gbps network card. We used the MPI framework to distribute VerDP across multiple machines. We reserved one machine for verification, which is relatively inexpensive; this left 31 machines available for proof generation.

For our experiments, we used a leaf group size of 2,048, the largest our EC2 instances could support without running out of memory. (Recall that this parameter, and thus the "width" of the map tiles, needs to be defined in advance by the curator and cannot be altered by the analyst.) We then generated four synthetic data sets for each query, with 4,096, 16,384, 32,768, and 63,488 rows, which corresponds to 2, 8, 16, and 31 map tiles. Recall that VerDP's privacy guarantee depends critically on the fact that the structure of the computation is independent of the actual data; hence, we could have gained no additional insights by using actual private data. Although some real-world data sets (e.g., the U.S. census data) are larger than our synthetic data sets, we believe that these experiments demonstrate the key trends.

7.3 Commitment generation

Before the curator can make the data set available to analysts, he must first generate a commitment to the data set, and publish it. To quantify the cost, we measured the time taken to generate commitments for various database sizes, as well as the size of the commitment itself.

As expected, generating the commitment is not expensive: we found that the time varied from 1 second for our smallest data set (4,096 rows) to 3.1 seconds for our largest data set (63,488 rows). The size of the commitment is 256 bits, independent of the size of the data set; recall that the commitment is generated through a hash tree, and only the final root-hash is committed to. These costs seem practical, even for a curator with modest resources – especially since they are incurred only once for each new data set.

7.4 Query compilation and EK generation

With the database and the commitment, the analyst can formulate and test queries in VFuzz. Once the analyst has finalized the set of queries, she sends the queries to the curator, who compiles them and generates an EK and VK for each tile, as well as the seeds for each noise generator. Since the curator has to independently recompile the VFuzz queries, it is important that compilation is relatively inexpensive.

Recall from Section 4.3 that a single VFuzz query can contain multiple "red" parts, depending on the number of `sample` calls, and that each part can consist of multiple map tiles (depending on the database size), a tree of reduce tiles, and a single noising tile that returns the final "sample".

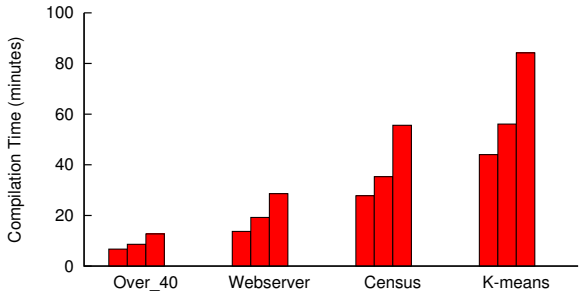

Figure 6. Compilation time for map tiles as a function of tile size, for (512, 1024, 2048) rows per tile.

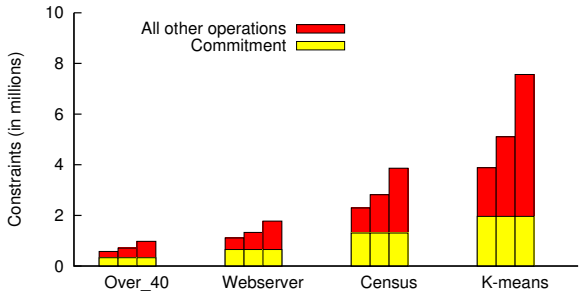

Figure 7. Constraints generated for map tiles with (512, 1024, 2048) rows per tile.

Our queries contain between one and six `sample` calls each. However, recall that compilation is only required for each *distinct* tile; since most map tiles (and *all* reduce and noising tiles) are identical, compilation time depends only on the width of the map tile, but *not* on the size of the data set.

Time: To estimate the burden on the curator, we benchmarked the time it took to compile our map, reduce, and noising tiles, and to generate the corresponding EKs and VKs; our results are shown in Figure 6. We can see that compilation and key generation is relatively inexpensive, taking at most 84 minutes for the largest query (k-means). The compiled binaries are also relatively small, taking at most 14 MB. Note that the curator *must* compile the binaries in order to produce the keys; the analyst can either download the compiled binaries from the curator or recompile them locally with the EK.

Complexity: We also counted the number of constraints that were generated during compilation, as a measure of complexity. Figure 7 shows the results for each of the map tiles. We separate out the number of constraints used for 1) commitment operations, and 2) actual computation. The figure shows that a large part of the overhead is in commitment operations. This is why small tile widths are inefficient.

To summarize, the work done by the curator is relatively inexpensive. Generating commitments is cheap and must be done only once per data set. The costs of compilation and key generation are nontrivial, but they are affordable as they do not grow with the size of the data set.

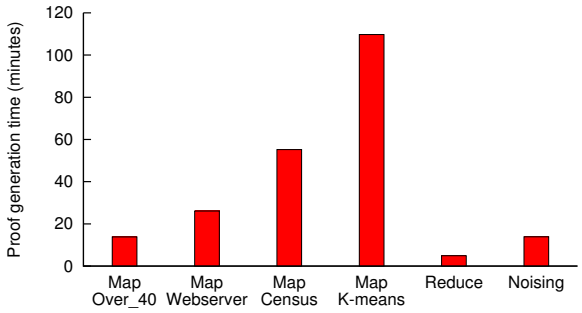

Figure 8. Time to generate proofs for map tiles of width 2,048, as well as the reduce and noising tiles.

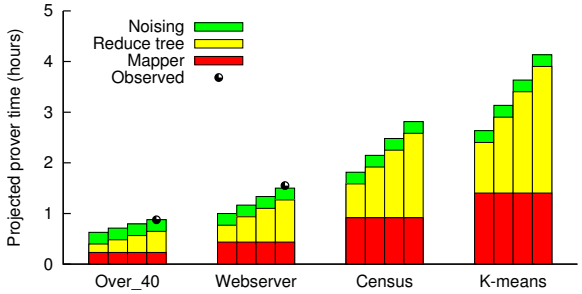

Figure 9. Projected time to generate proofs for databases of (8k, 16k, 32k, 62k) rows.

7.5 Query execution and proof generation

Once the analyst receives the compiled queries from the curator, she must run the resulting verifiable program to generate the results and proof, and then make the proof available to interested readers. This is the most computationally expensive step. To quantify the cost, we generated proofs for each of our four queries, using varying data set sizes.

Microbenchmarks: We benchmarked the map tiles for each of our four queries, using a tile width of 2,048 rows, as well as the reduce and noising tiles. (The `cg1.4xlarge` instances could not support map tiles with more rows, due to memory limitations.) Our results are shown in Figure 8. Proof generation times depend on the complexity of the query but are generally nontrivial: a proof for a k-means map tile takes almost two hours. However, recall from Section 5.2 that VerDP can scale by generating tile proofs in parallel on separate machines: all the map tiles and all the reduce tiles at the same level of the reduce tree can run simultaneously. As a result, the time per tile does not necessarily limit the overall size of the data set that can be supported.

Projected runtimes: For larger databases that require $k > 1$ map tiles, we estimate the end-to-end running time and cost. Since all the map tiles can be run in parallel when k machines are available, a data set of size $2048 * k$ can be run in the time it takes to run a single mapper. The reduce tiles, however, need to be run in at least $\log_2 k$ stages, since we need to build a binary tree until we reach a single value, which is then handed to a noising tile. Figure 9 shows our estimates;

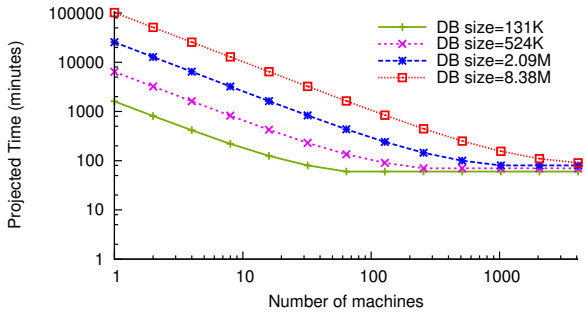

Figure 10. Estimated time to run the "over 40" query using a variable number of machines. Note log-log scale.

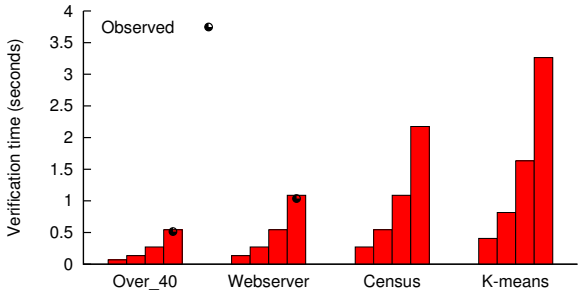

Figure 11. Projected time to verify complete programs for databases of (16k, 32k, 62k) rows on one machine.

doubling the size of the data set only adds a constant amount to the end-to-end proof generation time, although it does of course double the number of required machines.

Projected scaling: Figure 10 shows the projected runtime for different levels of parallelization. VerDP scales very well with the number of available machines; for instance, 32 machines can handle 524K rows in about 230 minutes. Eventually, scalability is limited by the dependencies between the tiles – e.g., map needs to run before reduce, and the different levels of the reduce tree need to run in sequence. Note that the depth of the reduce tree, and thus the amount of non-parallel work, grows logarithmically with the database size.

End-to-end simulation: To check our projected runtimes, we ran two end-to-end experiments, using the over-40 and weblog queries, data sets with 63,488 rows, and our 32 `cg1.4xlarge` EC2 instances. We measured the total time it took to generate each proof (including the coordination overhead from MPI). The results are overlaid on Figure 9 as individual data points; they are within 3.3% of our end-to-end results, which confirms the accuracy of our projections. We also note that, at $2 per instance hour at the time of writing, the two experiments cost $64 and $128, respectively, which should be affordable for many analysts.

Proof size: The largest proof we generated was 20 kB. The very small size is expected: a Pinocchio proof is only 288 bytes, and the commitments are 32 bytes each; each tile produces a single Pinocchio proof and up to three commitments, depending on the type of tile. Thus, a proof can easily be downloaded by readers.

7.6 Proof verification

When an interested reader wants to verify a published result, she must first obtain the commitment from the curator, as well as the query and the proof from the analyst. She must then recompile the query with VerDP and run the verifier. To quantify how long this last step would take, we verified each of the tiles we generated.

Figure 11 shows our projected verification times, based on the number of tiles and our measured cost of verifying each individual tile. We also verified our two end-to-end proved queries, and those are displayed in the graph. The measured results were 4.7% faster than our projections, confirming the accuracy of the latter.

We note that, during verification, tiles do not have dependencies on any other tiles, so verification could in principle be completely parallelized. But, at total run times below 3.5 seconds in all cases, this seems unnecessary – readers can easily perform them sequentially.

7.7 Summary

Our experiments show that VerDP can handle several realistic queries with plausible data set sizes. The time to generate proofs is nontrivial, but nevertheless seems practical, since proof generation is not an interactive task – it can be executed in the background, e.g., while the analyst is working on the final version of the paper.

VerDP imposes moderate overhead on the curator. This is important as a curator might be serving the data set to many different analysts. Appropriately, the bulk of the computation time is borne by the analysts. Since the proofs are small and can be verified within seconds, proof verification is feasible even for readers without fast network connections or powerful machines.

8. Related Work

Differential privacy: To the best of our knowledge, VerDP is the first system to provide both differential privacy and verifiable integrity. Most existing systems focus exclusively on privacy; for instance, Fuzz [21], PINQ [30], Airavat [40], GUPT [34], and CertiPriv [2] are all either designed to check or to enforce that a given query is differentially private. A few recent systems have considered richer threat models: for instance, there are some solutions [14, 33, 36] that can process queries over multiple databases whose curators do not trust each other, while others [10] can prevent malicious *subjects* from providing data that would distort the results. But none of these systems enable readers to verify that the execution was performed correctly.

Verifiable computation: Although there has been significant theoretical work on proof-based verifiable computation for quite some time (see [38, 43, 47] for surveys),

efforts to create working implementations have only begun recently. These efforts have taken a number of different approaches. One set of projects [11, 45, 46], derived from Goldwasser et al.'s interactive proofs [20], uses a complexity-theoretic protocol that does not require cryptography, making it very efficient for certain applications. But, its expressiveness is limited to straight-line programs. Another line of work [41–43, 47] combines the interactive arguments of Ishai et al. [29] with a compiler that supports program constructs such as branches, loops, and inequalities and an implementation that leverages GPU cryptography. Zaatar [41], the latest work in that series, exploits the constraint encoding of Gennaro et al. [19] for smaller, more efficient proofs. This encoding is also used by Pinocchio [38], which offers similar functionality to Zaatar while supporting proofs that are both non-interactive and zero-knowledge. Pantry [8], the system we use for VerDP, enables verifiable programs to make use of state that is only stored with the prover and not the verifier while supporting both the Zaatar and Pinocchio protocols. Recently, several promising works have enabled support for data-dependent loops via novel circuit representations [4, 5, 48]; a future version of VerDP could incorporate techniques from these systems to improve VFuzz's expressiveness.

Language-based zero-knowledge proofs: Several existing systems, including ZKPDL [31], ZQL [16], and ZØ [17], provide programming languages for zero-knowledge proofs. ZQL and ZØ are closest to our work: they enable zero-knowledge verifiable computations over private data for applications such as personalized loyalty cards and crowdsourced traffic statistics. Like VerDP, they provide compilers that convert programs written in a high-level language to a representation amenable to proofs, but VerDP provides a stronger privacy guarantee. ZQL and ZØ allow the programmer to designate which program variables are made public and which are kept private to the prover, but he or she has no way of determining how much private information the verifier might be able to infer from the public values. VerDP, the other hand, bounds these leaks using differential privacy.

9. Conclusion and Future Work

VerDP offers both strong certifiable privacy and verifiable integrity guarantees, which allows *any* reader to hold analysts accountable for their published results, even when they depend on private data. Thus, VerDP can help to strengthen reproducibility – one of the key principles of the scientific method – in cases where the original data is too sensitive to be shared widely. The costs of VerDP are largely the costs of verifiable computation, which recent advances by Pinocchio [38] and Pantry [8] have brought down to practical levels – particularly on the verification side, which is important in our setting. Even though the costs remain nontrivial, we are encouraged by the fact that VerDP is efficiently parallelizable, and it can, in principle, handle large data sets.

Acknowledgments

We thank our shepherd Frans Kaashoek and the anonymous reviewers for their comments and suggestions. This work was supported by NSF grants CNS-1065060 and CNS-1054229, as well as DARPA contract FA8650-11-C-7189.

References

[1] Barbaro, M., Zeller, T., and Hansell, S. A face is exposed for AOL searcher No. 4417749. *The New York Times* (August 9, 2006). http://www.nytimes.com/2006/08/09/technology/09aol.html.

[2] Barthe, G., Köpf, B., Olmedo, F., and Zanella Béguelin, S. Probabilistic relational reasoning for differential privacy. In *Proc. POPL* (2012).

[3] Bell, R. M., and Koren, Y. Lessons from the Netflix prize challenge. *SIGKDD Explor. Newsl. 9*, 2 (2007).

[4] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., and Virza, M. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proc. CRYPTO* (2013).

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

[6] Blake, H., Watt, H., and Winnett, R. Millions of surgery patients at risk in drug research fraud scandal. *The Telegraph* (March 3, 2011). http://www.telegraph.co.uk/health/8360667/Millions-of-surgery-patients-at-risk-in-drug-research-fraud-scandal.html.

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

[8] Braun, B., Feldman, A. J., Ren, Z., Setty, S., Blumberg, A. J., and Walfish, M. Verifying computations with state. In *Proc. SOSP* (2013).

[9] Chawla, S., Dwork, C., McSherry, F., Smith, A., and Wee, H. Toward privacy in public databases. In *Proc. TCC* (2005).

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

[11] Cormode, G., Mitzenmacher, M., and Thaler, J. Practical verified computation with streaming interactive proofs. In *Proc. ITCS* (2012).

[12] Deer, B. MMR doctor Andrew Wakefield fixed data on autism. *The Sunday Times* (February 8, 2009). http://www.thesundaytimes.co.uk/sto/public/news/article148992.ece.

[13] Dwork, C. Differential privacy: A survey of results. In *Proc. TAMC* (2008).

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

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

[16] Fournet, C., Kohlweiss, M., Danezis, G., and Luo, Z. ZQL: A compiler for privacy-preserving data processing. In *Proc. USENIX Security* (2013).

[17] Fredrikson, M., and Livshits, B. ZØ: An optimizing distributing zero-knowledge compiler. In *Proc. USENIX Security* (2014).

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

[19] Gennaro, R., Gentry, C., Parno, B., and Raykova, M. Quadratic span programs and succinct NIZKs without PCPs. In *Proc. EUROCRYPT* (2013).

[20] Goldwasser, S., Kalai, Y. T., and Rothblum, G. N. Delegating computation: Interactive proofs for muggles. In *Proc. STOC* (2008).

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

[22] Hawblitzel, C., Howell, J., Lorch, J. R., Narayan, A., Parno, B., Zhang, D., and Zill, B. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. OSDI* (2014).

[23] Herndon, T., Ash, M., and Pollin, R. Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. Working paper 322, Political Economy Research Institute, University of Massachusetts Amherst, 2013. http://www.peri.umass.edu/fileadmin/pdf/working_papers/working_papers_301-350/WP322.pdf.

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

[25] ICPSR Data Deposit at the University of Michigan. http://www.icpsr.umich.edu/icpsrweb/deposit/.

[26] Integrating Data for Analysis, Anonymization and SHaring. http://idash.ucsd.edu/.

[27] Interlandi, J. An unwelcome discovery. *The New York Times* (October 22, 2006). www.nytimes.com/2006/10/22/magazine/22sciencefraud.html.

[28] Integrated Public Use Microdata Series at the Minnesota Population Center. https://www.ipums.org/.

[29] Ishai, Y., Kushilevitz, E., and Ostrovsky, R. Efficient arguments without short PCPs. In *Proc. CCC* (2007).

[30] McSherry, F. Privacy Integrated Queries. In *Proc. SIGMOD* (2009).

[31] Meiklejohn, S., Erway, C. C., Küpçü, A., Hinkle, T., and Lysyanskaya, A. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *Proc. USENIX Security* (2010).

[32] Mironov, I. On significance of the least significant bits for differential privacy. In *Proc. CCS* (2012).

[33] Mironov, I., Pandey, O., Reingold, O., and Vadhan, S. Computational differential privacy. In *Proc. CRYPTO* (2009).

[34] Mohan, P., Thakurta, A., Shi, E., Song, D., and Culler, D. GUPT: Privacy preserving data analysis made easy. In *Proc. SIGMOD* (2012).

[35] Narayan, A., Feldman, A., Papadimitriou, A., and Haeberlen, A. Verifiable differential privacy. Tech. Rep. MS-CIS-15-05, Department of Computer and Information Science, University of Pennsylvania, Mar. 2015.

[36] Narayan, A., and Haeberlen, A. DJoin: Differentially private join queries over distributed databases. In *Proc. OSDI* (2012).

[37] Narayanan, A., and Shmatikov, V. Robust de-anonymization of large sparse datasets. In *Proc. S&P* (2008).

[38] Parno, B., Gentry, C., Howell, J., and Raykova, M. Pinocchio: Nearly practical verifiable computation. In *Proc. S&P* (2013).

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

[40] Roy, I., Setty, S., Kilzer, A., Shmatikov, V., and Witchel, E. Airavat: Security and privacy for MapReduce. In *Proc. NSDI* (2010).

[41] Setty, S., Braun, B., Vu, V., Blumberg, A. J., Parno, B., and Walfish, M. Resolving the conflict between generality and plausibility in verified computation. In *Proc. EuroSys* (2013).

[42] Setty, S., McPherson, R., Blumberg, A. J., and Walfish, M. Making argument systems for outsourced computation practical (sometimes). In *Proc. NDSS* (2012).

[43] Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A. J., and Walfish, M. Taking proof-based verified computation a few steps closer to practicality. In *Proc.USENIX Security* (2012).

[44] Sweeney, L. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10*, 05 (2002).

[45] Thaler, J. Time-optimal interactive proofs for circuit evaluation. In *Proc. CRYPTO* (2013).

[46] Thaler, J., Roberts, M., Mitzenmacher, M., and Pfister, H. Verifiable computation with massively parallel interactive proofs. In *Proc. HotCloud* (2012).

[47] Vu, V., Setty, S., Blumberg, A. J., and Walfish, M. A hybrid architecture for interactive verifiable computation. In *Proc. S&P* (2013).

[48] Wahby, R. S., Setty, S., Ren, Z., Blumberg, A. J., and Walfish, M. Efficient RAM and control flow in verifiable outsourced computation. Cryptology ePrint 2014/674, 2014.