DISTRIBUTED DIFFERENTIAL PRIVACY AND APPLICATIONS

Arjun Narayan

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2015

Andreas Haeberlen, Assistant Professor of Computer and Information Science
Supervisor of Dissertation

Lyle Ungar, Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee:
Boon Thau Loo, Associate Professor of Computer and Information Science
Zachary G. Ives, Professor of Computer and Information Science
Aaron Roth, Assistant Professor of Computer and Information Science
Micah Sherr, Assistant Professor of Computer Science, Georgetown University

DISTRIBUTED DIFFERENTIAL PRIVACY AND APPLICATIONS

Acknowledgments

First and foremost, I'd like to thank Andreas Haeberlen. I could not have had a better advisor. I'd also like to thank my excellent dissertation committee for their suggestions and generosity with their time: Boon Thau Loo, whose course I took in my first year and collaborated with, and who also gave me valuable career advice, Aaron Roth, who taught me a tremendous amount about differential privacy, Zack Ives, who taught an excellent databases course and provided valuable feedback from a different perspective, and Micah Sherr, for his early collaboration that helped me get my feet wet doing research, as well as his comments that improved this document.

The technical material in this thesis has been previously published. Fuzz was published in USENIX Security 2011 (Haeberlen, Pierce, and Narayan, 2011), DJoin in OSDI 2012 (Narayan and Haeberlen, 2012), and VerDP in Eurosys 2015 (Narayan, Papadimitriou, and Haeberlen, 2015). All that work was a group effort, and for that I thank my collaborators Jason Reed, Antonis Papadimitriou, Ariel Feldman, and Benjamin Pierce for their valuable contributions. In addition to everything else, Andreas was of course also a large part of all of those efforts.

At Penn I benefited greatly from the advice and guidance of wizened elders: Jason Reed got me bootstrapped and running, Sudipto Guha provided much-needed guidance, and Michael Greenberg—despite his best efforts not to—helped. My first year cohort was always there: Mukund Raghothaman, Abhishek Udupa, Mingchen Zhao. I shared many late-night debugging adventures with them. I hope that I offered some advice of use in turn to Justin Hsu, Ang Chen, and Antonis Papadimitriou. Even if I didn't, they very graciously pretended otherwise. The conference circuit was always there too: Mike Chow, Michael Z. Lee, Bill Jannen, and recurring roommate Malte Schwarzkopf helped me survive the stressful rites of passage of giving talks and presenting posters. Finally, Mike Felker helped ensure that the countless bureaucracies were handled behind the scenes.

Arno Dirks, Andrew Mahlstedt, Sudarshan Cadambi, my mother, and my sister

were always there for me. I would not have started a PhD without my father's guidance. Finally, Jennifer made the entire process worthwhile. Ernest, the world's best dog, ensured a healthy dissertation writing schedule, and that I didn't write more than a couple pages at a time before mandatory belly-rub breaks.

ABSTRACT

DISTRIBUTED DIFFERENTIAL PRIVACY AND APPLICATIONS

Arjun Narayan

Andreas Haeberlen

Recent growth in the size and scope of databases has resulted in more research into making productive use of this data. Unfortunately, a significant stumbling block which remains is protecting the privacy of the individuals that populate these datasets. As people spend more time connected to the Internet, and conduct more of their daily lives online, privacy becomes a more important consideration, just as the data becomes more useful for researchers, companies, and individuals. As a result, plenty of important information remains locked down and unavailable to honest researchers today, due to fears that data leakages will harm individuals.

Recent research in differential privacy opens a promising pathway to guarantee individual privacy while simultaneously making use of the data to answer useful queries. Differential privacy is a theory that provides provable information theoretic guarantees on what any answer may reveal about any single individual in the database. This approach has resulted in a flurry of recent research, presenting novel algorithms that can compute a rich class of computations in this setting.

In this dissertation, we focus on some real world challenges that arise when trying to provide differential privacy guarantees in the real world. We design and build runtimes that achieve the mathematical differential privacy guarantee in the face of three real world challenges: securing the runtimes against adversaries, enabling readers to verify that the answers are accurate, and dealing with data distributed across multiple domains.

Contents

List of Tables

List of Figures

1

Introduction

Recent growth in the size and scope of databases has resulted in more research into making productive use of this data. Unfortunately, a significant stumbling block which remains is protecting the *privacy* of individuals that populate these datasets. As people spend more time connected to the Internet, and conduct more of their daily lives online, privacy becomes a more important consideration, just as the very same data becomes potentially more useful for researchers, companies, and individuals. Today, plenty of important information is locked down and unavailable to honest researchers due to fears that data leakages will harm individuals. Despite the growth in useful datasets such as healthcare records and census records, these are generally not available for academic or scientific use without great restrictions and access controls. In some cases, as with healthcare records (CDC and HHS, 2003) or educational records (O'Donnell, 2002), there are laws restricting how this information may be shared with any third party, regardless of their good intentions or protections.

It is not just governments that collect large potentially useful datasets. Increas-

ingly, private companies are also stewards of large datasets. These companies wish to use datasets such as Netflix's movie ratings (Bell and Koren, 2007), Amazon's retail purchase histories (Linden, Smith, and York, 2003), and several companies that seek to provide better targeted advertising (Mayer and Mitchell, 2012). Nevertheless, sharing this data with researchers or other companies is not easy: it is unclear what private information may be inadvertently revealed by sharing a particular dataset. Even honest companies who spend lots of money and effort can get it wrong, by making incorrect assumptions about the power of their adversaries (Narayanan and Shmatikov, 2008). This uncertainty about the potential privacy losses creates an environment where data is seldom shared, even if there are large benefits from doing so.

Recent research in *differential privacy* (Dwork, McSherry, Nissim, and Smith, 2006b) gives us a promising way to *quantify* the privacy loss to individuals when sharing private information. Differential privacy provides *provable* information theoretic guarantees on what any answer may reveal about any single individual in the database. Unlike earlier work on data anonymization, which seeks to "scrub" a dataset clean of identifying information, differential privacy is a property of *queries*. In this setting, the private dataset is not revealed to the third party. Instead, a third party can pose questions, incrementally consuming a *privacy budget*. Once the budget is depleted, no more questions are answered. This framework allows researchers to make a strong case when asking for individuals to allow their data to be shared: they can precisely quantify what will be revealed about any individual, in the worst case situation. This powerful guarantee has resulted in a flurry of research (Dwork, 2008), presenting novel algorithms to compute a rich class of computations that provide differential privacy.

However, in practice, differential privacy is hard to guarantee. While the guarantee can be achieved by several theoretical algorithms, *executing* those algorithms safely requires sufficiently addressing additional concerns. First and foremost is *se-*

cure execution. If arbitrary (and potentially adversarial) queriers supply programs, we need to ensure that those queries are safe. Even if we can certify that their *outputs* are differentially private, the programs may exploit *side channels*, such as the time taken to run to completion. A second concern is the fact that private data is often dispersed across multiple administrative domains, and some important queries cannot be cleanly broken down into sub-queries that can be answered individually, and then post-processed into a final result. Third is the concern that differentially private answers are noised, and thus contain an element of plausible deniability *on the part of the database owner*. Thus, if there is the possibility of fraudulent answers, queriers have no way of knowing whether the answer was computed correctly, and the noise added honestly from a random distribution. For differential privacy to be usable in the real world, we need to build systems that sufficiently address these practical concerns.

1.1 Thesis

In this dissertation, we focus on real world scenarios that pose significant challenges when providing differential privacy guarantees in practice. A differential privacy system needs to address the following three needs:

- *Secure execution*: A runtime needs to be secure against malicious adversaries that attempt to breach the query system's guarantees via malicious queries that attempt to infer additional information via side channels such as execution time. Without this starting guarantee, any theoretical privacy guarantees on the output can be circumvented and rendered useless.
- *Distributed execution*: A runtime should be able to answer questions when the underlying data is distributed among multiple parties who may not necessarily trust each other to combine their datasets.

- *Verifiable answers*: Answers should be provably correct, and these proofs should be efficiently checkable. A proof of correctness that a given answer is differentially private should also not leak any additional private information.

1.2 Contributions

In this dissertation, we make the following contributions to address those concerns:

1. We present Fuzz, a differentially private database engine that provides *secure execution*, protecting against side channel and covert channel attacks.

   Our key insight is using a new primitive that we call predictable transactions - executing differentially private queries such that the execution takes a precise predetermined amount of time, regardless of the value of the final result.

2. We present DJoin, a differentially private database engine that can answer queries about private data that is spread across multiple different databases. DJoin can support many SQL-style queries, including joins on databases maintained by mutually untrusting entities.

   Our key insight is that some useful JOINs can be expressed as set intersections, for which there are existing private algorithms. We modify one such algorithm to make it *differentially* private, and rewrite our queries to use the underlying set intersection primitive for execution.

3. We present VerDP, a system for private data analysis that provides *both* strong integrity *and* strong differential privacy guarantees. This allows a querier to verify that a given differentially private answer was generated honestly, and without any errors in its calculation.

   The main challenge in verifying execution traces of a differentially private query processor is that the structure of the trace can leak information by itself, even if the verification is done by using a zero-knowledge proof. Our key insight is

that we can transform Fuzz programs to a variant where the circuit generated from the execution trace is also private, allowing us to safely release it, along with a zero-knowledge proof that it was executed honestly on the underlying dataset.

1.3 Outline

1. We begin in chapter 2 with some background material on differential privacy, as well as discussing related work on building differentially private systems.

2. We describe in chapter 3 the design of a system, Fuzz, that guarantees *secure execution* in the setting where the database and runtime is located at a single party.

3. We then describe in chapter 4 the design of DJoin, which allows for *Join* queries in the multiparty setting.

4. We describe in chapter 5 the design of VerDP, a system that gives zero-knowledge proofs of correct execution in addition to differentially private answers, allowing readers to verify the correctness of private results.

5. In chapter 6, we conclude with a discussion of potential future work.

2

Background and related work

2.1 Privacy

In many situations, participants in a computation care that their data will be kept private. This can be a nebulous and ill-defined requirement (Acquisti and Grossklags, 2007), and often depends on the specific use-case in question. In this dissertation I focus on one specific situation: where datasets are collected explicitly for the purpose of using the data in aggregation. This is typical of statistical research, where a single anecdote is anyway not scientifically relevant. However, this definition of privacy specifically ignores other settings, such as private outsourced computation (Atallah and Frikken, 2010) or transmitting private conversations (Garfinkel, 1995), where the term *privacy* is used in reference to hiding information from third parties entirely.

In this setting, the first and most basic approach to ensuring database privacy before releasing a dataset is *anonymization*. Anonymization is the process by which a dataset is stripped of explicitly identifying information , such as names and social security numbers. While necessary, this is, however, insufficient on its own

in the face of adversaries that have *auxiliary information* about a potential victim. This adversarial information can appear innocuous, making reasoning about it on a case-by-case basis infeasible. For instance, in a *linkage attack*, a publicly available non-anonymized database is linked back on an individual-by-individual level to the anonymized database, allowing adversaries to *deanonymize* the dataset. For example, Narayanan and Shmatikov (2008) deanonymized the anonymized Netflix dataset by linking it to the public IMDB dataset, as many viewers had rated some movies on both datasets. Consider an example user Alice, who likes watching movies. She watches some some set of movies A that she is comfortable being publicly linked to her identity. She rates these movies on a public website such as IMDB. There is also a second set of movies that she watches, B that she is embarrassed about. However, in her private Netflix account, she rates both A and B. If Netflix releases a deanonymized set of ratings, a linkage attack can link the common subset A between both accounts, which, in effect, means that her ratings for the set B are now also public.

Linkage attacks can be mitigated by only releasing *aggregate* statistics. A privacy criterion known as K-anonymity, first formalized by Sweeney (2002), is the condition that no single aggregate statistic is released unless at least k individuals are in the aggregation. This gives every user a crowd of $k-1$ others to hide in. This, however, has two flaws: it first results in a steep decrease in utility (Aggarwal, 2005) for high dimensional datasets, and second, it is *still* not completely secure: K-anonymity is not safe in the face of powerful adversaries. While a linkage attack would not work against a $k-$anonymized dataset, an adversary who knew the information of $k-1$ other individuals can still learn information about single users. A second attack is possible if the adversary controls the questions asked, and can ask multiple questions. Consider a database of private salary information of computer science professors at the University of Pennsylvania. An adversary could ask for the average salary of all the professors in August 2015. Since there are a large number of professors in the

department, this satisfies the $k-$anonymity rule even with a large k (which is presumably, more private). In September 2015, a new professor joins. The adversary asks the same question, and can thus calculate the new professor's exact salary. The flaw is that the meta-query (constructed by the difference of the two queries) has exactly 1 person in the set (breaking the k limit on any question). In general, reasoning about all possible meta-queries that are composed of answered queries grows exponentially. Thus, K-anonymization does not give us the strong privacy guarantees we desire.

Finally, privacy is not just a concern when entire datasets are released. while the Netflix dataset deanonymization was done on an anonymized dataset released for research purposes in improving their recommendation engine, just the recommendation engine itself can be used to break individual user's privacy. For instance, Calandrino, Kilzer, Narayanan, Felten, and Shmatikov (2011) use a moderate amount of auxiliary information to infer individual purchases based on the changes in what the engine recommends over time.

2.2 Differential Privacy

Instead of attempting to scrub a dataset until it is "perfectly anonymized", an impossible target (Dwork and Naor, 2008), we could try a second approach: releasing just the result of a computation based on the private data, without releasing the dataset itself. This is considerably more private, and, if done carefully, can leak almost no individual information at all. Nevertheless, this approach is still fallible: powerful adversaries with lots of auxiliary information can still use the final result to deduce information about individuals. A hypothetical adversary who knows everything about all users in the dataset except for Alice, can learn her information. This is not as far fetched as it sounds, if the adversary can ask some questions, as in the previous example with professor salaries, he can quickly launch a differential attack.

Differential Privacy is a stronger requirement: it information theoretically limits what any query says about *any* single individual in the private dataset, regardless of

their auxiliary knowledge. Differential privacy is not a property of databases, but a property of *queries*. The intuition behind differential privacy is that we bound how much the output can change if we change the data of a single individual in the database. Formally, if we encode an individual's data as a *single row* in the database, the property we desire is that for any two datasets d_1 and d_2 which differ in a single row, a function f over the range of outputs R is ε-differentially private if for all subsets S of R, it satisfies the following condition:

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

The above inequality means that any change in a single row results in at most a multiplicative change of e^{ε} in the probability of any output. Differential privacy does not *guarantee* that an adversary will not learn some private fact about you: instead, it guarantees that the differentially private results will not *raise the probability* of any adversary learning any individual's private data by more than the factor e^{ε}, which is $\approx 1+\varepsilon$ if $\varepsilon \ll 1$.

Methods for achieving differential privacy can be attractively simple–e.g., perturbing the true answer to a numeric query with carefully calibrated random noise. For example, the query "How many patients at this hospital are over the age of 40" is intuitively "almost safe": safe because it aggregates many individuals' information together, but only "almost" because, if an adversary happened to know the ages of every patient except John Doe, then answering this query exactly would give him certain knowledge of a fact about John. The differential privacy methodology rests on the observation that, if we add a small amount of random noise to this query's result, we still get a useful estimate of the true answer while obscuring the age of any single individual. By contrast, the query, "How many patients named John Doe are over the age of 40?" is plainly problematic, since the answer is very sensitive to the presence or absence of a single individual. Such a query cannot usefully be privatized: if we add enough noise to mostly obscure the contribution of John Doe's age,

there will be essentially no signal left.

A common way to achieve differential privacy for queries with numeric outputs is the *Laplace mechanism* (Dwork et al., 2006b), which works as follows: Suppose $\bar{q} : I \to R$ is a deterministic, real-valued function over the input data, and suppose $\bar{q}$ has a finite *sensitivity* s to changes in its input, i.e., $|\bar{q}(d_1) - \bar{q}(d_2)| \leq s$ for all similar databases $d_1, d_2 \in I$. Then $q := \bar{q} + Lap(\frac{s}{\varepsilon})$, i.e., the combination of $\bar{q}$ and a noise term drawn from a Laplace distribution with scale parameter $\frac{s}{\varepsilon}$, is ε-differentially private. This corresponds to the intuition that the more sensitive the query, and the stronger the desired guarantee, the more "noise" is needed to achieve that guarantee.

Adding noise degrades the utility of the output, as compared to releasing the unnoised deterministic answer itself, but if done carefully, we can limit the degradation to still useful levels. For example, in many statistical settings, there is already an implicit level of noise (due to sampling error) in the data, so researchers are usually prepared to deal with noise in the aggregate results anyway. Thus, for a small cost to utility, we get strong privacy guarantees. Differential privacy systems are parameterized by a tunable ε parameter, which controls *how private* the answer is. Setting this parameter is itself a complex task; Hsu, Gaboardi, Haeberlen, Khanna, Narayan, Pierce, and Roth (2014) build threat models to determine how to precisely set ε, while still achieving reasonable utility from the result.

2.2.1 Compositionality and privacy budgets

An important consequence of the definition of differential privacy is that composing a differentially private function with any other function that does not, itself, depend on the database yields a function that is again differentially private—that is, no amount of post-processing, even with unknown auxiliary information, can lessen the differential privacy guarantee. This allows us to reason about harmful effects of data release that might seem quite far removed from the function that is actually being computed.

Another important property of differential privacy is that its guarantee degrades gracefully under repeated application: a pair of two ε-differentially private functions is always, at worst, 2ε-differentially private, when taken together. This allows us to think of having a fixed "privacy budget" up front, which is slowly exhausted as queries are answered: if our privacy budget is ε, we may feel free to independently answer k queries, where the i^{th} query is ε_i-differentially private and $\sum_i \varepsilon_i \leq \varepsilon$, without fear that the aggregation of these k queries will violate ε-differential privacy.

2.2.2 Function sensitivity

The central idea in proofs of differential privacy is to bound the *sensitivity* of queries to small changes in their inputs. Sensitivity is a kind of continuity property; a function of low sensitivity maps nearby inputs to nearby outputs.

Sensitivity is relevant to differential privacy because the amount of noise required to make a deterministic query differentially private is proportional to its sensitivity. For example, the sensitivity of the two age queries discussed above is 1: adding or removing one patient's records from the hospital database can change the true value of each query by at most 1. This means that we should add the *same* amount of noise to *"How many patients at this hospital are over the age of* 40*?"* as to *"How many patients named John Doe are over the age of* 40*?"* This may appear counter-intuitive, but it achieves the right goal: the privacy of single individuals is protected to exactly the same degree in both cases. What differs is the *usefulness* of the results: knowing the answer to the first query with, say, a typical error margin of ± 100 could still be valuable if there are thousands of patients, whereas knowing the answer to the second query (which can only be zero or one) ± 100 is useless. We might try making the second query more useful by scaling its answer up numerically: *"Is John Doe over* 40*? If yes, then* 1000*, else 0."* But this scaled query now has a sensitivity of 1000, not 1, and so 1000 times the noise must be added, blocking our attempt to violate privacy.

2.2.3 The Distributed Setting

This privacy guarantee on the output of the process, while involving considerable effort to guarantee, is still only one part of the story. In the single-party setting it is sufficient (as long as the single party who sees the private data is trusted, and safe from side channel attacks). However, if the computation is distributed among several administrative domains, we have to additionally consider how the distributed computation itself leaks between the domains: if Alice has trusted Bob with her private data, she may not authorize Bob to share her data with Carol in order to jointly compute some statistical results for David. Thus, our goal is to build systems that allow Bob to give provable privacy guarantees to Alice, while still performing distributed computations with Carol that have meaningful use. Thus, these privacy guarantees are *with respect to* Carol *and* David.

To avoid this case-by-case reasoning, a best-case criterion introduced in McGregor, Mironov, Pitassi, Reingold, Talwar, and Vadhan (2010) would be that the entire *transcript* of messages in a distributed computation, as well as the final result, is differentially private. This is the strongest possible guarantee, but results in degraded *utility*: we could potentially achieve better by allowing for distributed databases to communicate using cryptographically secure messages, or participate in secure multiparty computation protocols. While this is not information theoretically private, this is still a reasonable assumption in practice. Thus, our guarantees are weakened such that the transcript and output is only *computationally differentially private (Mironov, Pandey, Reingold, and Vadhan, 2009)*.

2.3 Related Work

In this section, we briefly summarize some related systems that provide differential privacy, in both the single party setting, and the distributed setting.

2.3.1 The Single Database Setting:

The first system that is most relevant is PINQ (McSherry, 2009). PINQ is a database engine in the client-server setting, which accepts queries from clients written in a privacy aware version of the LINQ language. PINQ is the most general purpose language: it contains a rich set of operators on the underlying data, like joins and group-bys, and supports arbitrary lambda functions in conjunction with these operators. PINQ thus allows for a rich set of queries, but as we will discuss in Chapter 3, does not provide for *secure execution* in the face of adversarial queriers who wish to exploit *side channels* in order to break the differential privacy guarantees.

The second system that is most relevant is Airavat (Roy, Setty, Kilzer, Shmatikov, and Witchel, 2010). Airavat is a differentially private query engine that executes queries with a MapReduce structure. Querier provided Mappers functions are restricted to observing a single database row, and have a system enforced maximum sensitivity. Airavat implements some trusted reducers, which in combination with the querier-provided Mappers, can provide differentially private results.

2.3.2 The Horizontally Partitioned Distributed Setting:

In this setting, the data is sharded across multiple curators, but each curator holds a subset of the rows, with the exact same columns. Systems can only execute queries that can be broken down into sub-queries executed over each shard, and then aggregated separately.

PDDP, by Chen, Reznichenko, Francis, and Gehrke (2012), is a system for differentially private web-analytics in the horizontally partitioned setting. Analysts wish to observe browsing behaviors of users in aggregate, but users want their individual private data protected from the analysts. PDDP's solution is to use an intermediate *proxy*, which sits between the (potentially multiple) analysts and the clients (which live on the individual users' computers). The analysts receive differentially private answers to their queries. PDDP requires that the proxy be trusted to add noise to

the answers, and not collude with the analysts. In return, by following this protocol, individuals are guaranteed that the analysts do not learn much about any single one of them.

SplitX, by Chen, Akkus, and Francis (2013), is an evolution of the PDDP design, and focuses on the same setting of analysts and distributed horizontally-partitioned clients. However, instead of a single proxy it relies upon three separate parties: two *mixes* and an *aggregator*. By splitting the responsibilities for privacy between three parties, it uses one-time pads to get much better performance than PDDP. However, in exchange, the system is reliant upon an additional assumption that no two of the three parties collude.

"Peers for Privacy" or P4P, by Duan, Canny, and Zhan (2010), is a system designed to execute machine learning workloads in a distributed private fashion. Users have private vectors, which are collected to form a matrix. Users use a secret sharing scheme to break up their data into two separate secrets, which are sent to two separate servers. The servers aggregate this information. When they combine the data, only the aggregated data is available to the servers. The servers can then execute any arbitrary function over the aggregated matrix. The motivation for P4P is to perform private singular value decompositions (SVDs) of the user data matrix.

2.3.3 The General Distributed Setting:

A different, more general setting, concerns data that is not just horizontally partitioned. Here, different users may have different columns of the dataset, in addition to different subsets of rows.

In this setting, one important primitive required is *distributed noise generation*. If many users wish to collaboratively aggregate their results privately, they need a protocol by which they can aggregate their results and add noise to it. Since for privacy purposes noise only needs to be added once, if every user adds noise independently at random, this results in excess noise. Secure multiparty computation

(SMC) (Yao, 1982) allows users to collaboratively execute a shared circuit, such that only the final result is revealed, and every user's individual input (and intermediate stage of the computation) is kept secret. Dwork, Kenthapadi, McSherry, Mironov, and Naor (2006a) presents an efficient circuit that adds noise. This is crucial, as SMC is computationally very expensive to run, and requires small circuits to be feasible. Critically, SMC can compute *any* function, thus freeing users from the restrictions of the horizontal partitioning setting. Once a final result is computed, the distributed noise generation circuit is used to add Laplace noise to the final result, before it is released.

Finally, Shi, Chan, Rieffel, Chow, and Song (2011a), details the design of a system for "Private Stream Aggregation", or (PSA). In this setting, users also wish to collaboratively generate a differentially private aggregation of their inputs. However, this setting assumes a stronger threat model than SplitX or PDDP: users do not have a single (or multiple) servers that they trust, even partially. Unlike Dwork et al. (2006a), where users execute a secure multiparty computation synchronously over a network, in PSA, users wish to execute results asynchronously, after an initial setup phase. In particular, users wish to execute a large number of aggregations. This is motivated by the setting of "time series data": for example, if users had smart meters but want their privacy protected, they would consent to differentially private aggregations of their data, but the utility provider would like periodic meter readings that are unbounded in time.

3

Securing Differentially Private Runtimes

3.1 Introduction

Early work on differentially private data analysis relied on manual proofs by privacy experts that the answers to particular queries were safe to release (McSherry and Mironov, 2009); today, systems like PINQ (McSherry, 2009) and Airavat (Roy et al., 2010) can perform differentially private data analysis automatically, without needing a human expert in the loop.

Airavat and PINQ go beyond just certifying queries by the data owner as differentially private; they are explicitly designed to support *untrusted* queries over private databases. In this model, a third party is permitted to submit arbitrary queries over the database, but the data owner imposes a privacy budget that limits the amount of information the third party can obtain about any individual whose data is in the database. The system analyzes each new query to determine its potential privacy cost and allows it to run only if the remaining balance on the privacy budget is sufficiently high. This mode of operation is attractive for many scenarios; for example, Netflix

could give researchers access to its database of movie ratings via such a query interface and still give strong privacy assurances to customers. An adversarial querier could not, for instance, obtain an accurate answer to the query "*Has John Doe watched any adult movies?*" because the cost of such a query would exceed any reasonable privacy budget.

However, Airavat and PINQ both contain vulnerabilities that can be exploited by an adversary to extract private information through covert channels.[1] The reason is that these systems rely on the assumption that the querier can observe *only* the result of the query, and nothing else. In practice, however, the querier is also able to observe other effects of his query, such the time it takes to complete. Such observations can be exploited to mount a covert-channel attack. To continue with our earlier example, the adversary might run a query that always returns zero as its result but that takes one hour to complete if John Doe has watched adult movies and less than a second otherwise. Both Airavat and PINQ would consider the output of such a query to be safe because it does not depend on the contents of the private database at all. However, the adversary can still learn with perfect certainty whether John Doe has watched adult movies—a blatant violation of differential privacy. PINQ's prototype implementation also permits global variables to be used as covert channels to leak private information during query execution.

Covert channels have plagued computer systems for many years (Lampson, 1973; Wray, 1991; Askarov, Zhang, and Myers, 2010; Shroff and Smith, 2008; Agat, 2000; Kang, Moskowitz, and Lee, 1996; Hu, 1991, etc.), and they are notoriously difficult to avoid (Crosby, Wallach, and Riedi, 2009). However, they are particularly devastating in a system that is designed to enforce differential privacy: if a channel allows the adversary to learn even a single bit of private information, the differential privacy guarantees are already broken! Thus, differential privacy puts particularly high demands on a defense against covert channels; merely limiting the bandwidth

[1] The designers of these systems were aware of these covert channels, and each addresses them to some extent. See Sections 3.3.5 and 3.3.6.

of the channels is not enough.

Fortunately, the untrusted-query scenario has two features that make a solution feasible. First, there is no need to allow the querier direct access to the machine that hosts the database; he can be forced to submit queries and receive results over the network. This rules out difficult channels such as power consumption (Kocher, Jaffe, and Jun, 1999) and electromagnetic radiation (Gandolfi, Mourtel, and Olivier, 2001; Quisquater and Samyde, 2001), essentially leaving the adversary with just two channels: the privacy budget and the query completion time.

Our key insight is that, in this specific scenario, these two channels can be closed *completely* through a combination of two techniques. The budget channel can be closed by using program analysis to statically determine the privacy cost of each query. Thus, the deduction from the privacy budget is independent of the database contents. The external timing channel can be closed by a) breaking each query into "microqueries" that operate on a single database row at a time, and by b) enforcing that each microquery takes a fixed amount of time. (If necessary, the microquery is aborted and a *default value* is returned. In the context of differential privacy, this is safe—and does not open another channel—because the privacy cost of the default values is already included in the privacy cost of the query.) Thus, we can obtain strong privacy assurances even if the adversary can pose arbitrary queries and can observe all the (remotely measurable) channels that are possible in our model.

We present the design of Fuzz, a system that implements this defense. Fuzz uses a type system from Reed and Pierce (2010) to statically infer the privacy cost of arbitrary queries written in a special programming language, and it uses a novel primitive called *predictable transactions* to ensure that a potentially adversarial computation completes within a specific time or returns a default value. We have built and evaluated a proof-of-concept implementation of Fuzz based on the Caml Light runtime system (Leroy, 1990; Leroy and Doligez, 2004). Our results show that Fuzz effectively closes all known remotely exploitable channels, at the expense of a higher

query completion time.

Implementing predictable transactions is challenging in practice: Fuzz must be able to abort an arbitrary and potentially adversarial computation by a specified deadline, even if the adversary is actively trying to cause the deadline to be missed, and must ensure that—whether or not the computation is aborted—it leaves no lingering traces that can measurably affect the program's overall execution time (garbage in the heap, VM pages that must later be freed by the OS, etc).

Nevertheless, we show that, across a variety of adversarial queries that exploit different attack strategies, our implementation exhibits extremely small variation in completion time—on the order of the time required to handle a single timer interrupt. This variation is so small that it is difficult to measure even on the machine itself. Thus, it would be useless to a remote attacker, who would have to measure it across a wide-area network using the limited number of trials that the privacy budget permits.

In summary, we make the following contributions in this chapter:

1. a detailed analysis of several classes of covert-channel attacks and a discussion of which are feasible in PINQ and Airavat (Section 3.3);
2. an analysis of the space of potential solutions (3.4);
3. a concrete design for one specific solution, based on default values and predictable transactions (3.5+3.6);
4. a proof-of-concept implementation of our design (3.7); and
5. an experimental evaluation (3.8).

We close with a discussion of related work and a few concluding thoughts.

3.2 Background

3.2.1 Programming with privacy

Early work on differential privacy has mostly focused on specific algorithms rather than general, compositional mechanisms: given a particular algorithm, we prove by hand that it is differentially private. Most of the time, this does not require much ingenuity—just applying known techniques—but even so, this approach doesn't scale well because it demands that each new algorithm be certified by a skilled, trusted human. A better approach is to automate this certification process with a programming language in which every well-typed program is guaranteed to be differentially private. Then (untrusted) non-experts can write as many different algorithms as they like, and the database administrator can rely on the language to ensure that privacy is not being violated.

Systems are beginning to be available that implement such languages—notably Privacy Integrated Queries (PINQ) (McSherry, 2009) and Airavat (Roy et al., 2010). PINQ is an embedded extension of C# that tracks the privacy impact of variety of relational algebra operations on database tables, as well as certain forms of query composition. Airavat integrates differential privacy into a distributed, Java-based MapReduce framework.

3.2.2 Processing model

Although PINQ and Airavat differ in many particulars, they embody essentially the same basic processing model, which we also follow in the Fuzz system described below. A *query* in each of these systems can be viewed as consisting of one or more *mapping* operations that process individual records in the database, together with some *reducing* code that combines the results of the mapping operations without directly looking at the database. When a query is submitted, the system verifies that it is ε_i-differentially private, deducts ε_i from the total privacy budget ε associated

with the database, and—if ε remains nonzero—returns the query result. (Note that, in this model, we account for the possibility of collusion between adversaries by associating the privacy budget with the *database* and not with individual queriers. Thus, once the budget is exhausted, we must throw away the database and never answer any more queries.) We call the mapping operations *microqueries* and the rest of the code the *macroquery*.

Airavat implements a simple version of this model: a query consists of a sequence of chained microqueries ("mappers" in Airavat terminology) plus a selection from among a fixed set of macroqueries ("reducers"). The mappers are the only untrusted code: the reducers are part of the trusted base. When a query is submitted, the adversary must also declare the expected numerical range of its outputs, which amounts (since its input is a single record of the database) to stating its sensitivity. If the actual output ever falls outside of the declared range, it is clipped—in essence, the declared sensitivity is enforced by the system. From the declared sensitivity, Airavat can calculate how much noise must be added to the reducer's results to achieve ε-differential privacy.

In PINQ, macroqueries are written in LINQ, a SQL-like declarative language, which can be embedded in otherwise unconstrained C# programs. Microqueries can be general C# computations (optionally constrained by a checker method called `Purify`; see Section 3.3.5).

3.3 Attacks on differential privacy

Naturally, database administrators may be nervous about offering adversaries the opportunity to run arbitrary queries against their raw data. They will need strong assurances that such adversarial queries not only play by the rules of differential privacy but also have no *indirect* means of improperly leaking private information about individuals in the database. Unfortunately, this is not currently the case: while the authors of both PINQ and Airavat have anticipated the possibility of covert-

channel attacks and have implemented either a partial defense (Airavat) or hooks for adding one (PINQ), both systems remain vulnerable to a range of attacks, as we now demonstrate.

3.3.1 Threat model

It is well known that covert channels are essentially impossible to eliminate if we allow the adversary to run other processes on the same computer that runs the query. Even if these other processes have no access to the database and cannot communicate directly with the query process, there are just too many ways for the query process to perturb local conditions in ways that can be measured fairly accurately if the observer is this close—e.g., processor usage, disk activity, cache pollution, etc. However, if we assume that the adversary is on the other end of a network connection, we have a much better chance of success. This is fortunate, since the demands of the situation are very strong. It is not enough to limit leakage to a low bandwidth or a small number of bits: even *one bit* is too much if that bit is the answer to *Does John Doe watch adult movies?*

We therefore assume that the database and associated query system are hosted on a private, secure machine. The adversary does not have physical access to this machine or its immediate environment (so that there is no way to measure its power usage, etc.) and can only communicate with it over a network. The adversary submits arbitrary queries to the system over the network. The system executes each query (if it determines that doing so is safe) and returns the answer over the network. The system also maintains a privacy budget for the database as a whole, and it refuses to answer any more queries once the budget is exhausted.

This threat model is shared by all differentially private query systems (PINQ, Airavat, and our Fuzz system), and its assumptions seem reasonable in practice. Essentially, it gives the adversary three pieces of information: (1) the actual answer to their query (a number, histogram, etc.), if any, (2) the *time* that the response arrives

on their end of the network connection, and (3) the system's decision whether to execute their query or refuse because doing so would exceed the available privacy budget. However, this threat model still provides plenty of room for attacks on privacy. We will see that, unless appropriate steps are taken, both the decision whether or not to execute a query and the execution time itself can be used as channels to leak private information. In essence, both the query's finishing time and the fact that it is accepted or refused are *results* that the system is giving back to the adversary, and we need to consider whether the combination of *all* results—not just the query's numerical answer—is differentially private. Moreover, we will see, for PINQ, some ways that a malicious query may cause the actual *answer* to not be differentially private.

3.3.2 Timing attacks

Under the constraints of the above threat model, the easiest way for a query to send a bit to the adversary is by simply pausing for a long time (by entering an infinite loop, computing factorial of a million, etc.) when a certain condition is detected in the private data, as illustrated (in PINQ-like pseudocode) in Figure 3.1. The macroquery adds together the results of running the microquery on each row of the database (always 0) and finally adds some random noise to the total. Since almost all of the microquery instances finish very quickly, the distribution of query execution times observed by the adversary will change significantly when an embarrassing record exists in the database—a violation of differential privacy.

A simple "microquery timeout" will not solve this problem, for at least two reasons. First, the adversary can also signal the condition by causing the query to take an unusually *small* amount of time. The simple way to do this is to create an exception condition that aborts the entire query. If this is blocked (e.g., by trapping an exception in a microquery and replacing it with a default result just for that single microquery), the adversary can instead make all microqueries take a uniformly longish time (say, exactly two milliseconds) except when they detect the condition,

```
noisy sum, foreach r in db, of {
  if embarrassing(r) {
    pause for 1 second
  };
  return 0
}
```

Figure 3.1: Timing attack example

```
found = false;
noisy sum, foreach r in db, of {
  if (found) then { return 1 }
  if embarrassing(r) then {
    found = true;
    return 1
  } else { return 0 }
}
```

Figure 3.2: State attack example

in which case they terminate immediately. If the adversary happens to know exactly how many records are in the database, this leaks one bit. Second, the adversary can defeat a simple "microquery timeout" by causing side-effects in the microquery that will slow down the macroquery or other microqueries—for example, by allocating lots of memory to trigger garbage collection in the macroquery. We discuss this issue in more detail below.

3.3.3 State attacks

A different class of attacks involves using a channel *between* microqueries, such as a global variable, to break differential privacy of the result, as illustrated in Figure 3.2. This time, the result of each microquery is either 0 or 1, depending on whether *any previous* microquery detected an embarrassing record. Since, in general, the embarrassing record will not be the last one in the database, this greatly magnifies

```
noisy sum, foreach r in db, of {
  if embarrassing(r) then {
    run sub-query that uses
      a lot of privacy budget
  } else {
    return 0
  }
}
```

Figure 3.3: Privacy budget attack example

	Constant execution time Database size public ε-differential privacy	**Variable execution time** Database size private (ε, δ)-differential privacy
Static enforcement	Exact timing analysis	Time bound analysis Time noise
Dynamic enforcement	Timeouts Rounding up	Timeouts Time noise

Table 3.1: Four approaches to the timing-channel problem.

the contribution of this one record to the result, again violating differential privacy.

3.3.4 Privacy budget attack

A related form of attack uses the query processor's decision whether to publicize the result of a query as a channel for leaking private data, relying on the fact that this decision can be influenced by actions of the query that in turn depend on private data. This idea can be applied to systems that use a *dynamic* analysis to determine the 'privacy cost' of a query, i.e., the amount that must be subtracted from the privacy budget before the result can be returned to the querier. As illustrated in Figure 3.3, the attack consists of looking for an embarrassing record and, when it is found, invoking some sub-query that will use up a bit of the remaining privacy budget. Once the outer query returns, the adversary simply checks how much the privacy budget has decreased.

3.3.5 Case study: PINQ

We have verified that the current PINQ implementation (version 0.1.1, released 08/18/09, available from (pinq)) is vulnerable to all of the above attacks. To demonstrate the vulnerabilities, we have written three example programs, each based on the test harness that comes with PINQ.

The original test harness computes several differentially private statistics on a given text file, including the number of lines that contain a semicolon. When the program starts, it first reads the text file and creates a database whose rows each contain one line of text. Then it selects all the rows that contain a semicolon, using microqueries with a boolean predicate p, and finally performs a noisy count on the resulting set of rows.

Our attacks are implemented by changing the predicate p so that it produces some observable side-effects when the input file contains a certain string s. For the timing attack, we changed p so that, when invoked on a line that contains s, p performs an expensive computation that takes several seconds and cannot be optimized out. For the state attack, we added a static variable that is incremented by p when it discovers s, and we write the (un-noised) value of this variable to the console at the end. For the budget attack, we added a different static variable that contains a reference to the database; when s is found, p computes a noisy count of the number of rows in the database, which decreases the privacy budget.

The possibility of such attacks is acknowledged in the PINQ paper (McSherry, 2009), and the PINQ implementation does contain hooks for an expression rewriter (called `Purify` in (McSherry, 2009)) that is invoked on all user-supplied expressions and could potentially change or remove code that causes side-effects. However, such a rewriter is not provided; indeed, the PINQ downloads page contains an explicit warning that the code is not hardened or secured and should not be used 'in the wild.'

We conjecture that implementing a reliable `Purify` will be far from trivial. Avoid-

ing the privacy budget attack will probably be easiest: every function that might consume privacy budget could be wrapped with a check that raises an exception if it is called from inside a running microquery (i.e., with a PINQ operation already on the call stack); this exception could then be turned into a default result for the microquery. State attacks are more difficult: since microqueries in PINQ are arbitrary bits of C#, it seems the choices are either to execute them on a modified virtual machine that detects writes to global state (as Airavat does), or else to create a small domain-specific language for writing microqueries that avoids global updates by design (as we do in Fuzz). Addressing timing attacks will require deeper changes to PINQ: the issues and available solutions are precisely the ones we study in this chapter.

3.3.6 Case study: Airavat

Because Airavat calculates sensitivity and deducts the required amount from the privacy budget before query execution begins, it is inherently safe from privacy budget attacks. However, Airavat's mechanism for preventing state attacks permits a related vulnerability. To prevent microqueries from communicating via static variables, Airavat runs microqueries on a modified JVM; if a microquery ever attempts to modify a static variable, an exception is thrown and the whole query is marked "not differentially private." Unfortunately, the adversary can now observe whether the system gives them the result at the end of query execution or says, "Sorry, that's not differentially private." A better alternative would be to abort just the microquery, return, a default result, and allow the remainder of the query to run to completion.

In its published form, Airavat is also vulnerable to timing attacks. Its authors acknowledge this weakness (Roy et al., 2010) but counter that the bandwidth of the channel it creates is very low. This, we agree, may make it tolerable in some contexts, e.g., with "mostly trusted" queriers that might be careless but will not write malicious queries that intentionally attempt to reveal specific targeted secrets.

We understand that Airavat may soon be enhanced to add timeouts to microquery executions [Shmatikov, personal communication, July 2010]; the implementation techniques described below should be useful in this effort.

3.4 Defending against timing attacks

State and privacy budget attacks can (and must) be addressed by designing the query language so that they are impossible. Timing attacks require more work, and this will be our concern for the remainder of the chapter.

3.4.1 Four approaches to the problem

There are two basic strategies. One is to ensure that a given query takes very close to the *same* amount of time for *all possible* databases (of a given size—see below), so that the adversary can learn nothing from observing the time it takes the query result to arrive. The other is to treat time as an additional output of the query, and to limit the amount of information the adversary can gain using the same mechanisms (sensitivity analysis and appropriate perturbation) that are used for data outputs.[2] In either approach, we can either obtain the information about running time statically (by analyzing the program before running it) or enforce limits dynamically (e.g., by using timeouts). This gives us the four possibilities shown in Table 3.1.

The solutions in the right-hand column provide somewhat weaker privacy guarantees than those on the left. In order to properly "noise" a resource like time, we must have the ability to both increase *and decrease* its consumption. While we can clearly increase execution time by adding a delay, we cannot easily decrease it. We can mitigate this problem by adding a default delay T; thus, we can add "time noise" $v \geq -T$ by delaying for $T + v$ at the end of each query. Nevertheless, since noise distributions guaranteeing differential privacy have unbounded support (i.e., $P(v) > 0$

[2] Note that the sensitivity analysis would have to account for interdependencies between a query's execution time and its output value, which is far from trivial.

for all v), there is always a possibility that $v < -T$, in which case we cannot complete the computation. Thus, ε-differential privacy seems impossible in practice; all we can hope for is the slightly weaker property of (ε, δ)-differential privacy (Dwork et al., 2006a), where δ is a bound on the maximum *additive* (not multiplicative) difference between the probability of any given query output with and without a particular row in the input.

On the other hand, in the constant-time solutions (left column), the size of the database becomes public knowledge, since, except for the most trivial queries, execution time depends on the size of the database. In practice, this is probably a reasonable concession. In the case of the variable-time solutions (right column), the size of the database does not need to be published.

The static solutions (top row) are attractive in principle, but they depend on a static analysis of time sensitivity—something that has proved challenging except for very simple, inexpressive programming languages. We therefore concentrate on the bottom row as being a more realistic target for a feasible implementation. In this row, we choose one column to explore further: the "constant execution time" alternative, where we try to make each microquery take as close as possible to exactly the same amount of time. (The "variable execution time" column is in principle feasible and also deserves exploration; we believe similar mechanisms will be required.)

3.4.2 Default values

The approach we explore in the rest of this chapter is to dynamically ensure that each microquery m takes the exact same amount of time T. If the microquery takes less time to execute, we delay it and only return its result after T. If the microquery has executed for time T without returning a result, we abort it. However, aborting the enclosing *macro*query is not an option because this would leak information to an adversarial querier. Instead, our approach is to have the microquery return a *default value* d in this case.

To avoid privacy leaks through the default value, d must not itself depend on the contents of the database. In Fuzz, a static value for d is included with the query. Also, for reasons that will become clear in Section 3.4.4, d should fall within the range of the microquery m.

3.4.3 Do default values decrease utility?

When the microquery for a row r times out while answering a non-adversarial query, the utility of the query's overall result almost inevitably degrades. After all, the result no longer incorporates the intended contribution of r or any other row whose microquery has timed out, but rather uses the default value for each such microquery. However, a non-adversarial querier can always avoid the inclusion of any default values by choosing a sufficiently high timeout. If the timeouts are chosen properly, *timeouts should never occur while answering non-adversarial queries*. Thus, the only querier who experiences degraded utility is the adversary.

The question, then, becomes how to choose the timeout values. One possible method is as follows. The querier is supplied with a reference implementation of the query processor that additionally outputs the maximum processing time T_{max} for each microquery. The querier can then (locally) test his queries on arbitrary databases of his own construction and thus infer a reasonable time bound. The querier then adds a small safety margin and uses, say, $1.1 \cdot T_{max}$ as the timeout for his query. He then submits the query to the actual query processor, to be run on the private database.

3.4.4 Do default values create privacy leaks?

At first glance, it may appear that default values are replacing one evil with another: they seem to plug the timing channel at the expense of introducing a data channel. However, this is not the case: as long as the timeouts are applied at the microquery level (as opposed to imposing a timeout on the whole query), differential privacy is

preserved, for the following reason.

First, recall that Fuzz is designed to ensure that the completion time of a query depends only on the size of the database, but not its contents. Since we have assumed that the size of the database is public, and since our threat model rules out all the other channels, the only remaining way in which private information could 'leak' is through the (noised) data that the query returns.

Now, recall that the type system Fuzz implements is based on the type system from (Reed and Pierce, 2010). As described in (Reed and Pierce, 2010), this type system ensures that all programs that type-check are differentially private. This is achieved by inferring an upper bound on the program's sensitivity to small changes in its inputs—specifically, a change to an individual database row.

Fuzz extends the type system from (Reed and Pierce, 2010) with microquery timeouts on `map` and `split`, but, crucially, timeouts do not increase the sensitivity of these two functions. The reason is that the sensitivity of `map` and `split` depends on the range of values that the microquery can return. Since the default value is taken from the range of values that the microquery can *already* return in the absence of timeouts, the addition of timeouts does not increase this range, and thus does not increase the sensitivity either.

Of course, running a query on a given database with and without timeouts (or with shorter vs. longer timeouts) can yield very different results. Suppose we have a database b and a function with microqueries that, without timeouts, produces an output o when it is run on b. If we now add a very short microquery timeout, we can easily cause *all* the microqueries to abort and return their default value, and the resulting output for the same database D can be dramatically different from o. However, this does not mean that differential privacy is violated. Recall from Chapter 2 that the differential privacy guarantee makes a statement about running *the same query* on two databases b and b' that differ in *exactly one row* r. If we run a query with timeouts on both b and b', the only microquery that could behave dif-

Primitive	Arguments	Return value
`map db f T d`	Database `db`, function `f`, timeout `T`, default value `d`	Database
`split db p T`	Database `db`, boolean predicate `p`, timeout `T`	Two databases
`count db`	Database `db`	Noised $\lvert$`db`$\rvert$
`sum db`	Database `db`	Noised $\sum_i$ `db`$_i$

Table 3.2: Critical primitives in the Fuzz language

ferently is the one on row r. All the other microqueries start in the same state for both databases, so their behavior will be exactly the same—they will either time out on both b and b', or on neither.

3.5 The Fuzz system

Next, we present the design of the Fuzz system, which represents one specific point (the lower left quadrant) in the solution space from Table 3.1. This point is a good first step because it works with existing programming-language technology and is relatively easy to implement.

3.5.1 Overview

Fuzz consists of three main components: a simple *programming language*, a *type checker*, and a *predictable query processor*. The programming language rules out channels based on global state or side effects, simply by not supporting any primitives that could produce either. The type checker rules out budget-based channels by statically checking queries before they are executed and rejecting any query that cannot be guaranteed to complete with the available balance. Finally, the predictable query processor closes timing-based channels by ensuring that each microquery terminates after very close to *exactly* a specified amount of time. Figure 3.4 illustrates our approach.

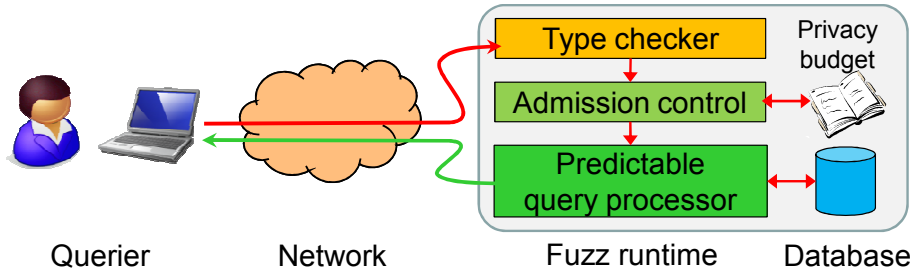


Figure 3.4: Scenario. Queries are first type-checked by Fuzz and then executed in predictable time.

3.5.2 Language and type system

Fuzz queries are written in a simple functional programming language whose functionality is roughly comparable to PINQ. The Fuzz language contains a special type *db* for databases, which is not a valid return type of any query. We say that a primitive is *critical* if it takes *db* as an argument. Our language ensures that critical primitives either return other values of type *db* (and nothing else) or add noise to all of their return values. Fuzz determines the correct amount of noise to add by using the sensitivity analysis and type system from (Reed and Pierce, 2010).

Fuzz currently supports four critical primitives (Table 3.2): `map` applies a function f to each row in one database and returns the results in another database; `split` applies a boolean predicate p to each row in a database and returns two databases, one with all rows r for which $p(r) = \text{TRUE}$ and the other with the rest; `count` returns the (noised) number of rows in a database; and `sum` returns the (noised) sum of all the rows. `sum`'s type ensures that it can only be applied to databases with numeric rows.

3.5.3 Predictable query processor

To close timing channels, the query processor must ensure that all critical primitives take a predictable amount of time that depends only on the size of the database. This

is trivial for `sum` and `count`. However, `map` and `split` involve arbitrary microqueries, and it can be difficult to statically analyze how much time these will take.

To avoid the need for such an analysis, Fuzz instead relies on *predictable transactions*. A predictable transaction is a primitive P-TRANS(λ, a, T, d), where λ is a function, a an argument, T a timeout, and d a default value. P-TRANS takes *exactly* time T, and returns $\lambda(a)$ if λ terminates within time T, or d otherwise. Note that an implementation of P-TRANS may have to (a) add a delay if λ terminates early, and (b) abort λ slightly *before* T expires to ensure that any resources allocated by λ can be released in time. In Section 3.6, we describe two approaches to implementing P-TRANS in practice.

When evaluating `map` or `split`, Fuzz invokes P-TRANS for each microquery, using the specified timeout T and—in the case of `map`—the specified default value (`split` has an implicit default of TRUE).

All values of type *db* internally have representations of the same size, i.e., they consume the same amount of memory and (conceptually) have the same number of rows as the original database. If necessary, they are padded with dummy rows. For example, if the original database has 1,000 rows and consumes 1 MB of memory, the two databases returned by a `split` both consume 1 MB, and an invocation of `map` on either of them will invoke 1,000 microqueries—though of course the results of microqueries on dummy rows will be discarded.

3.5.4 How Fuzz protects privacy

We now briefly summarize how Fuzz protects against covert channels. First, the only observations a querier can make that depend on the contents of the database are the completion time of the query and its return value. This is because of (a) our threat model from Section 3.3.1, (b) the fact that the language contains no primitives with side-effects, such as mutating global state, and (c) the fact that the type system rules out abnormal termination.

Second, the return value of the query is differentially private. Since *db* is not a valid return type and critical operations return only values of type *db* or else appropriately noised values (based on the sensitivity that has been statically inferred (Reed and Pierce, 2010)), the return value cannot depend on non-noised values from the database directly. Also, the language does not contain any primitives for observing side-effects within the query, such as memory consumption or the current wallclock time. The only time-related primitives are the timeouts on the microqueries; these have a sensitivity of 1 because (a) each microquery operates on only one row from the database at a time, and (b) microqueries have no access to global state and therefore cannot communicate with one another. Thus, if we add or remove one individual's data from the database, this affects only one row, so this can only cause one more (or less) microquery to time out and add a default result to the output.

Third, the completion time of a query depends only on the size of the database (which we assumed to be public) and data that has already been noised. To see why, consider that the only operations that have access to non-noised data are the microqueries, for which Fuzz enforces a constant runtime (by aborting or padding them to their timeout), and that values of type *db* cannot affect the control flow directly, only indirectly through return values of critical operations, which are noised. It is perfectly OK for the completion time of a query to depend on noised data, since such data is safe to release and could even have been returned to the querier directly.

In summary, Fuzz is designed to ensure that everything observable by the querier—whether directly through the data channel or indirectly through the timing channel—either does not depend on the contents of the database or has been noised appropriately.

3.6 Implementation strategies

In this section, we describe the abstract requirements for implementing predictable transactions, and we propose two concrete implementation strategies: one for newly

designed runtimes (3.6.2) and one for retrofitting Fuzz into an existing runtime (3.6.3). Naturally, we expect the former to be more efficient and the latter to be easier to implement.

3.6.1 Requirements

To implement p-trans(λ, a, T, d), the following three properties need to hold for the language runtime:

- **Isolation:** $\lambda(a)$ can be executed without interfering with the succeeding computation in any way, apart from contributing its return value.
- **Preemptability:** The execution of $\lambda(a)$ can be aborted at any time, or at most within some time bound Δ_a;
- **Bounded deallocation:** At any point during the execution of $\lambda(a)$, there is a upper bound Δ_d on the time needed to deallocate all resources allocated so far by $\lambda(a)$.

If these requirements hold, we can implement p-trans by running $\lambda(a)$ in isolation and setting a timer to $T - \Delta_a - \Delta_d$ (which must be updated when Δ_d changes due to new allocations). If the timer fires, we can abort λ and deallocate its resources without overrunning the overall timeout T. After a final delay to reach T exactly, we can return either the result of $\lambda(a)$ if we have it, or d otherwise.

3.6.2 White-box approach

If we design a new language runtime from scratch, or if we are willing to make extensive changes to an existing runtime, we can achieve isolation and preemptability by avoiding global variables that could be left in an inconsistent state when a microquery is aborted, as well as any termination of the microquery that does not correctly return the default value. Thus, it becomes possible to abort a microquery simply by performing a `longjmp` or its equivalent.

Regarding bounded deallocation, we expect that the key resource in most cases will be memory. It is possible to design the memory allocator in such a way that the memory allocated by a microquery can be deallocated in *constant* time. For example, we can divert the allocator from its usual allocation pool while a microquery is in progress, and instead allocate memory from a special region dedicated to microqueries. If the microquery takes arguments and returns results by value rather than by reference, objects in the main heap cannot acquire references to this region, so it is safe to summarily deallocate the entire region when the microquery aborts or terminates.

3.6.3 Black-box approach

The first strategy assumes a fairly deep understanding of how all primitive operations of the language are implemented, and how they interact with the allocator and each other. If we are working with an existing runtime system, it may be hard to be sure that the entire rest of the state of memory outside the microquery allocation region has been restored to its original state after a microquery finishes; for example, if we use any off-the-shelf library functions, they may have local buffers or other global state through which information can leak.

In this case, we can still ensure isolation and preemptability by leveraging operating system support, e.g., by farming out microqueries to a separate process, which can then be destroyed at any time without interfering with the state of the main runtime. Bounded deallocation can be achieved if we know an upper bound on the amount of time the operating system needs to destroy a process.

3.7 Proof-of-concept implementation

Next, we describe our proof-of-concept implementation of Fuzz. Our implementation does not execute Fuzz programs directly; rather, we implemented a front-end that accepts Fuzz programs, typechecks them, and then (if successful) translates

them into Caml programs. Thus, we did not need to implement an entire language runtime from scratch; it was sufficient to implement a library with Fuzz-specific primitives like `map` and `split`, and to extend an existing runtime with support for predictable transactions. We chose Caml because it is similar enough to Fuzz to make the translation relatively straightforward.

3.7.1 Background: Caml Light

Our implementation is based on Caml Light (Leroy, 1990; Leroy and Doligez, 2004) version 0.75, a stable and lightweight implementation of Caml. Here, we briefly describe only the aspects of Caml Light that are relevant for our discussion of Fuzz. For a detailed description of Caml Light, please see (Leroy, 1990).

In Caml Light, Caml code is first compiled into bytecode for an abstract machine called ZAM (the ZINC abstract machine); this bytecode is then executed on a runtime that implements the ZAM. Because of this architecture, the actual ZAM runtime is relatively simple: it mainly consists of an interpreter for the ZAM instructions and some code for I/O, memory management, and garbage collection.

The state of the ZAM consists of a code pointer, a register holding the current environment, an accumulator, two stacks (an argument stack and a return stack) and the heap. The heap is divided into two zones: a fixed-size 'young' zone and a variable-size 'old' zone. Most objects are initially allocated in the young zone; when this zone fills up, a 'minor' garbage collection copies any objects that remain active into the old zone. This was originally done to reduce the frequency of 'major' garbage collection runs (since most objects are short-lived, their space can be reclaimed very quickly), but it is also very convenient for Fuzz, as we shall see below.

Note that Fuzz uses the ZAM runtime to run only programs that it has previously translated from Fuzz programs. Thus, we can safely ignore features of the ZAM runtime (such as reference cells) that Fuzz does not use. Our threat model assumes that the adversary can submit only Fuzz programs, so he or she is unable to access

any of these features.

3.7.2 Bounded deallocation

When a microquery times out, Fuzz must be able, within a bounded amount of time, to release all of the resources the microquery may have allocated. To this end, our implementation performs a minor collection at the beginning of each macroquery, which clears the young zone of the heap, and it confines any additional memory allocations during microqueries to the young zone. Thus, we can simply discard the *entire* young zone after each microquery, which requires only a single instruction. If the microquery completes normally (without a timeout), it writes its result into a special fixed-size buffer that is not part of young zone. If this buffer is empty after the microquery or contains only a partial result, the macroquery uses the default value instead.

Discarding the entire young zone is safe because, after a microquery, there cannot be any outside references to objects in that zone. Any new memory allocations must be in the young zone, any new values on the stacks are discarded as well, and the only objects in the old zone that could be modified in place are reference cells, which translated Fuzz programs cannot use. Note that discarding the young zone is faster than a minor collection, so this particular modification (which is only possible for Fuzz programs, not for arbitrary Caml programs) actually results in a speedup.

3.7.3 Preemptability

Fuzz must be able to preempt a running microquery after a specified time, with high precision. To this end, our implementation creates a second thread that continuously spins on the CPU's timestamp counter (TSC).[3] When a microquery is started, the interpreter sets a shared variable to the time at which the preemption should occur; when that point is reached, the second thread sends a signal to the

[3] There are many other ways of implementing preemptions, such as periodic TSC checks in the interpreter loop, or using the CPU's performance counters.

interpreter thread. To prevent the two threads from slowing each other down, each is pinned to a different CPU core. If the microquery terminates before the timeout, it simply spins until the preemption occurs.

Preemptions can occur at arbitrary points in the runtime code. To avoid inconsistencies, our implementation checkpoints all mutable state before each microquery; when the signal is raised, it uses `longjmp` to return to the macroquery and then restores the runtime state from the checkpoint. We exclude from the checkpoint any state that either is immutable or is discarded anyway – including both zones of the heap and any existing values on the stacks. This leaves just a handful of variables, such as the ZAM's stack pointers and the code pointer.

3.7.4 Isolation

Fuzz must ensure that a microquery cannot interfere with the rest of the computation in any way, other than contributing its return value. In the previous two sections, we have already seen that the states of the ZAM runtime before and after a microquery are logically equivalent, since any changes (other than the result value) are either discarded or rolled back. To avoid direct timing interference between microqueries, Fuzz also pads the runtime of the preemption code to $\Delta_a + \Delta_d$. However, Fuzz must also avoid indirect timing interference through the garbage collector, or from the rest of the system.

Fuzz prevents data-dependent invocations of the garbage collector by padding all database rows to consume the same amount of memory, and by padding all database objects to have the same number of rows. For databases that result from a `split`, Fuzz adds an appropriate number of dummy rows that consume memory and computation time but do not contribute to the result. Fuzz also disables the garbage collector during microqueries; if a microquery attempts to allocate more space than is available in the young zone of the heap, Fuzz stops it and forces it to time out. Thus, from the perspective of the macroquery (and the garbage collector), memory

usage does not depend on un-noised values from the database.

To prevent page faults and context switches, Fuzz preallocates and pins all of its memory pages, and it assigns itself a real-time scheduling priority. In our experiments, this was sufficient to control the timing variations to within a a few microseconds.

3.7.5 Implementation effort

Altogether, we added or modified 6,256 lines of code, including 4,887 lines of C++ for the typechecker/translator, 1,119 lines of C++ and Caml code for our implementation of predictable transactions, 186 lines of C++ for benchmarking support, and 64 lines of Fuzz code for common library functions. For comparison, the entire Caml Light codebase consists of 29,984 lines of code. This supports our claim that Fuzz can be retrofitted into existing runtimes.

3.7.6 Limitations

Despite all our precautions, some potential sources of variability remain. For example, our current implementation does not freeze or flush the CPU's caches (since instructions like `wbinvd` are not available from user level), and it is designed to run on a commodity Linux kernel. We believe that these sources would be difficult to exploit because the adversary cannot control the memory layout or force the runtime to invoke system calls; also, any exploitable variation would have to be large enough to cause the $\Delta_a + \Delta_d$ padding to be overrun. An implementation with at least some kernel support could remove some or all of these sources, and thus use a less conservative padding.

3.8 Evaluation

Our evaluation has two primary goals. First, we need to demonstrate that Fuzz is *practical*, in the sense that it is sufficiently fast and expressive to process realistic

Name	Type	LoC	Inspired by
kmeans	Clustering	119	(Blum et al., 2005)
census	Aggregation	50	(Chawla et al., 2005)
weblog	Histogram	45	(Dwork et al., 2006b)

Table 3.3: Examples of non-adversarial Fuzz queries.

queries. Second, we need to demonstrate that our Fuzz implementation is *effective*, i.e., that it prevents all the covert-channel attacks that are possible in our threat model (Section 3.3.1).

3.8.1 Non-adversarial queries

To demonstrate that Fuzz is powerful enough to support useful queries, we implemented three example queries that were motivated in prior work (Dwork et al., 2006b; Blum, Dwork, McSherry, and Nissim, 2005; Chawla, Dwork, McSherry, Smith, and Wee, 2005). The **weblog** query is intended to run on the log of an Apache web server; it computes a histogram of the number of web requests that came from specific subnets. The **kmeans** query clusters a set of points and returns the three cluster centers, and the **census** query runs on census data and reports the income differential between men and women.

Table 3.3 reports the lines of code needed for each query. The queries are small because programmers only need to specify the actual data processing; parsing and I/O are handled by Fuzz. Also, the queries use a small library of generic primitives, such as lists and a `fold` operator, that consists of 64 lines written directly in the Fuzz language. Note that Fuzz can automatically certify queries as differentially private and perform sensitivity analysis during typechecking, so even non-experts can easily write differentially private queries.

3.8.2 Experimental setup

To evaluate the performance and effectiveness of Fuzz, we performed experiments using a setup consistent with our model from Section 3.3.1. We installed Fuzz on a dedicated machine, a Dell Optiplex 780 with a 3.06 Ghz Intel Core 2 Duo E7600 processor and 4 GB of memory. The machine was running a 32-bit Ubuntu Linux 11.04 with a 2.6.38-8 kernel. For our timing measurements, we used the CPU's timestamp counter, which is cycle-accurate. To minimize interference, we disabled CPU power management and the flush daemon, we kept all mutable data in a ramdisk and mounted all other file systems read-only, and we terminated all other processes on the machine, leaving Fuzz as the only running process (recall our assumption that the machine is dedicated to Fuzz). As discussed in Section 3.7.6, there are sources of timing variability that we could not disable, such as the periodic timer interrupt, which takes about 3 μs to handle in this setup, but these cannot be influenced by an adversary, so they merely add noise to the query completion time without leaking information. The padding time, which corresponds to $\Delta_a + \Delta_d$, was set to 10 μs; this setting was chosen to be the highest preemption latency we observed, plus a generous safety margin.

To estimate the overhead of our implementation, we also prepared a version of the three translated Fuzz queries that can run on the original Caml Light runtime. Since the original runtime does not support p-trans or a fixed-size memory representation for databases, this required small modifications to the Caml code; for example, the modified queries invoke microqueries without any timeouts, and they keep the database in ordinary Caml lists. These modifications do not affect the data output of the queries. We used the modified Caml code only for experiments with the original Caml Light runtime; all other experiments directly use the Caml code that is output by the Fuzz front-end.

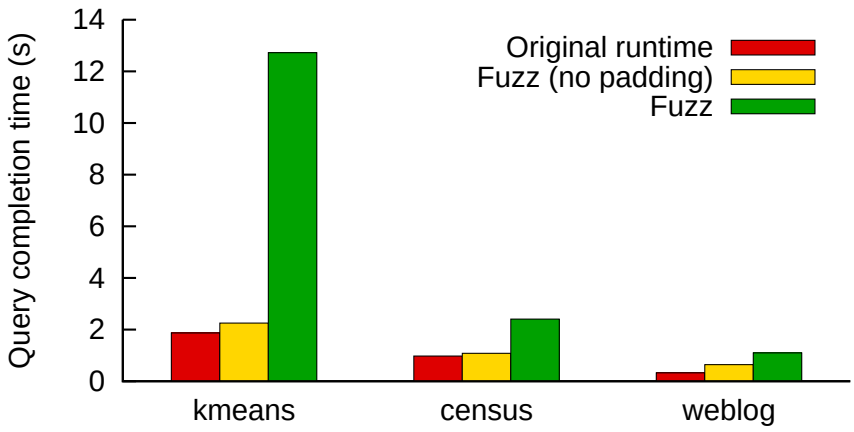


Figure 3.5: Performance for non-adversarial queries.

3.8.3 Macrobenchmarks

To estimate the performance of Fuzz, we ran each of the example queries from Table 3.3 over a synthetic dataset and measured the query completion time. Using synthetic data rather than real private data does not affect our measurements because, by design, the completion time does not depend on the contents of the database. However, the data *format* was based on realistic data—specifically, the weblog input was based on an Apache server log and the census input was based on U.S. census data from (Hettich and Bay, 2015). The synthetic database in each case had 10,000 rows. We set the microquery timeouts for each `map` and `split` by first running the query over example data with timeouts and padding disabled, measuring the maximum time taken by any of the `map` or `split`'s microqueries, and then setting the timeout to be 10% above that. We verified that no timeouts occurred during our measurements.

Figure 3.5 shows the query completion time for three different configurations: the original Caml Light runtime, the Fuzz runtime with both timeouts and padding disabled, and the Fuzz runtime with all features enabled. As expected, Fuzz takes more time to complete the queries than the original runtime; for our three queries, the slowdown was between 2.5x (census) and 6.8x (kmeans). However, in abso-

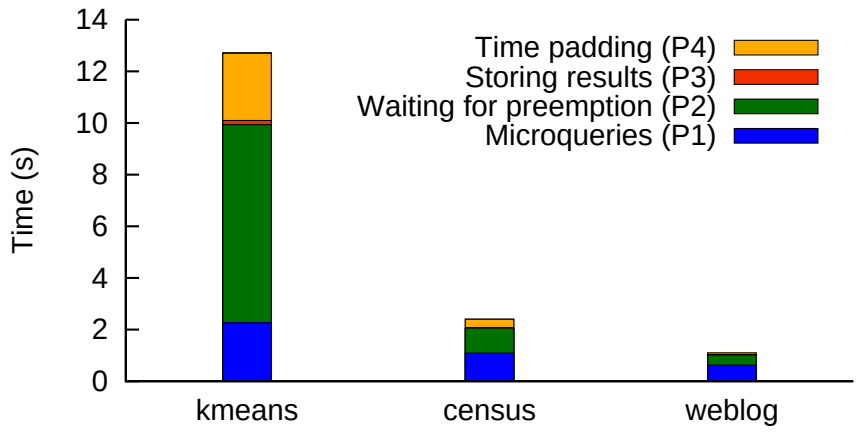


Figure 3.6: Time spent in different phases of query processing.

lute terms, the completion times were not unreasonable: the most expensive query (kmeans) took 12.7s to complete, which seems low enough to be practical.

Figure 3.5 also shows that, with timeouts and padding disabled, Fuzz's performance is roughly comparable to that of the original Caml Light runtime. This is not an apples-to-apples comparison; for example, the fixed-size memory representation for databases costs performance, whereas erasing the young zone after each microquery is actually faster than garbage-collecting it. Nevertheless, the numbers suggest that most of the overhead comes from padding and timeouts. Next, we examine this in more detail.

3.8.4 Microbenchmarks

To get a better picture of what factors influence the performance of our implementation, we added instrumentation in such a way that query time can be attributed to one of the following five phases:

- **P1:** Computation performed by a microquery;
- **P2:** Waiting for the preemption when a microquery completes early;
- **P3:** Preemption handling, storing results, restoring checkpoints, and loading

the next row;

- **P4:** Padding the time of the preemption handler to $\Delta_a + \Delta_d$; and
- **P5:** Computation performed by the macroquery.

Figure 3.6 shows our results (we omit the time P5 taken by the macroquery because it was below 0.2% of the total for all queries). As already suggested by the previous section, the majority of the time is spent in either the waiting or the padding phase. This may seem rather conservative at first, but recall that the completion time of even a non-adversarial microquery can vary with the row it is processing; the timeout needs to be sufficient for the longest query with high probability. Timeout handling, deallocation, checkpointing, and storing the results takes comparatively little time.

Note that the overhead for the kmeans query is considerably higher than for the others. This is because kmeans repeatedly uses `split` to partition the database – specifically, to map each point to the nearest of the three cluster centers. Since our proof-of-concept implementation is not keeping track of the fact that the union of the three partitions contains exactly the N rows in the original database, it must conservatively assume that *each* partition might contain *all* the N rows. Thus, functions that operate on the partitions are padded to $3 \cdot N$ times the timeout, when in fact N times would be sufficient. This could be avoided by extending Fuzz with a suitable operator, e.g., a GroupBy as in PINQ.

3.8.5 Adversarial queries

As explained in Section 3.5.4, Fuzz rules out state attacks and privacy budget attacks by design, and it prevents timing attacks by enforcing that each microquery takes precisely the time specified by its timeout. This last point cannot be perfectly achieved by a practical implementation running on real hardware; we need to quantify how close our implementation comes to this goal.

To this end, we implemented five adversarial queries, exploiting different variants

Attack type	Camllight runtime (unprotected)			Fuzz runtime (protected)		
	Hit	Miss	|Hit−Miss|	Hit	Miss	|Hit-Miss|
Memory allocation	1.961 s	0.317 s	1.644 s	1.101 s	1.101 s	<1 μs
Garbage creation	1.567 s	0.318 s	1.249 s	1.101 s	1.101 s	<1 μs
Artificial delay	1.621 s	0.318 s	1.303 s	1.101 s	1.101 s	<1 μs
Early termination	26.378 s	26.384 s	0.006 s	1.101 s	1.101 s	<1 μs
Artificial delay	2.168 s	0.897 s	1.271 s	2.404 s	2.404 s	<1 μs

Table 3.4: Effect of various attacks without and with predictable transactions. Each adversarial query tries to vary its completion time based on whether some specific individual is in the database. We show the total macroquery processing times when the individual is present (hit) and absent (miss), as well as the differences.

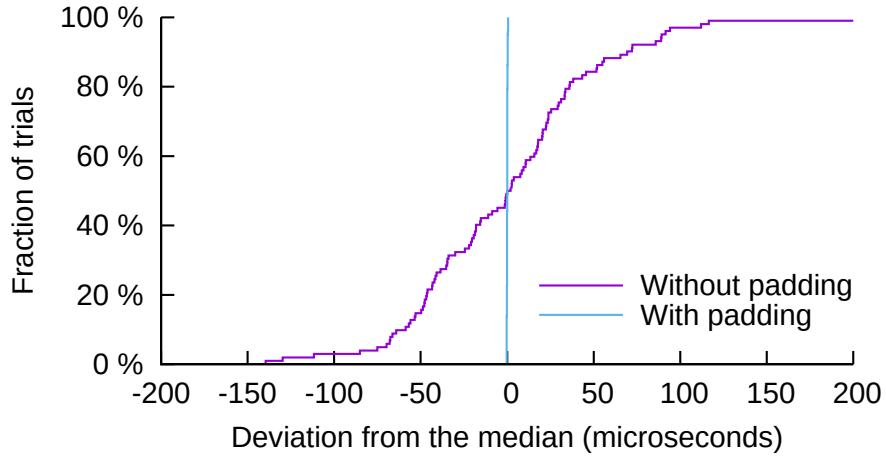


Figure 3.7: Variation of completion time for the weblog-delay query.

of the attacks from Section 3.3 to try to vary the completion time based on whether or not some specific individual is in the database:

- **weblog-delay** adds an artificial delay in each microquery that finds a match;
- **weblog-term** adds an artificial delay *except* when a microquery finds a match;
- **weblog-mem** consumes a lot of memory when a matching individual is found;
- **weblog-gc** creates a lot of garbage on the heap by repeatedly allocating and releasing memory;
- **census-delay** looks for a particular known person in the database and adds a

timing delay if their income is above a specified threshold.

We ran each query on two versions of the corresponding database: one that contains the individual (Hit) and another that does not (Miss). To demonstrate the effectiveness of these attacks on an unprotected system, we first performed the experiment with Fuzz runtime and then repeated it with the original Caml Light runtime. This gives us four configurations per query. We ran 100 trials for each configuration, after a warm-up phase of two trials to ensure that the Fuzz binary and the database were in the file system caches.

Figure 3.7 shows how the completion times varied across the 100 trials, using the weblog-delay query with the Miss database as an example. With the original runtime, the completion times varied by approximately ± 150 μs around the median. With the Fuzz runtime, the completion times are extremely stable: the difference between maximum and minimum was <1 μs. The results for the other queries were similar, indicating that Fuzz's padding mechanism successfully masks internal variations between trials. Hence, we only report median values here.

Table 3.4 shows our results for the different configurations. We make the following three observations. First, the attacks are very effective when protections are disabled. For four out of the five queries, the completion times for the Hit cases were at least one second different from the completion times for the Miss cases, so an adversarial querier could easily have distinguished between the two cases and thus learned with certainty whether or not the individual was in the database. We could have achieved even higher differences simply by changing the queries. For weblog-term, the difference was only a few milliseconds; the reason is that, in order to change the completion time of the query by one second through early termination, the adversary would have had to make *each* microquery take at least one second, so the overall query would have taken a conspicuously long time – in this case, nearly three hours.

Second, the attacks cease to be effective in Fuzz. In each case, the difference

between Hit and Miss is so small we could not even reliably measure it locally on the machine (for comparison, handling a timer interrupt requires about 3 μs, and one hundred of these are triggered every second, limiting the achievable accuracy), much less across a wide-area network, using the small number of trials that the privacy budget allows.

Third, the completion times are higher when protections are enabled. This is consistent with our earlier observations from Section 3.8.3.

3.8.6 Summary

Our results show that Fuzz is effective: it eliminates state and budget channels by design, and narrows the timing channel to a point where it ceases to be useful to an adversary. Query completion times remain practical but are substantially higher than in an unprotected system.

3.9 Related work on side and covert channel attacks

Floating Point attacks: Since the publication of Fuzz in 2010, there have been two attacks that have subverted the Fuzz prototype's privacy guarantees, and both exploit flaws in the floating point computations done towards the end of processing a Fuzz query that samples from the Laplace distribution.

The first work, Mironov (2012), broke the privacy guarantees of multiple systems, including PINQ, Airavat, and Fuzz, by exploiting the *least significant bits* of the output result. Since the set of available floating point numbers are not uniformly distributed between any two numbers, the standard algorithm for sampling from the Laplace distribution (which all the systems including Fuzz used) has a skewed distribution of *lower order bits* in its result, which Mironov (2012) use to reveal some private information. This leak is easily fixed by rounding up the final noised result to a fixed level of precision determined in advance, on the same level of precision as the unnoised query result.

A second attack, Andrysco, Kohlbrenner, Mowery, Jhala, Lerner, and Shacham (2015), exploits timing variations in the floating point operations when executed on an x86 processor in the same final sampling step from the Laplace distribution, to reverse engineer the unnoised private answer. This attack only works under conditions outside our threat model—the timing variation is so fine it requires the attacker to be observing on the same machine, while we assume in Fuzz that the attacker is behind a network. However, it can be completely closed by using predictable transactions for the final post-processing step as well.

Differentially private systems: While there is a considerable body of work on the theory of differential privacy (Dwork, 2006, 2008, 2009) and on differentially private data analysis (McSherry, 2009; Roy et al., 2010), except for the papers on Airavat (Roy et al., 2010) and PINQ (McSherry, 2009), none of these papers discuss covert-channel attacks by adversarial queriers. The PINQ paper briefly mentions certain security issues, such as exceptions and non-termination; Airavat discusses timing channels, but, as we have shown in Section 3.3.5, its defense is not fully effective. This chapter complements existing work by providing a practical defense against covert-channel attacks, which could be applied to existing systems.

Covert channels: Covert channels have plagued systems for decades (Lampson, 1973; Wray, 1991), and they are notoriously hard to avoid in general. Fuzz is a domain-specific solution; it only addresses differentially private query processing, but it can give strong assurances in this specific setting.

A variety of defenses against covert channels have been suggested. Most related to this chapter is the work on external timing channels. The bandwidth of external timing channels can be reduced, e.g., by adding random delays (Kang et al., 1996; Hu, 1991) or by time quantization (Askarov et al., 2010). However, to guarantee differential privacy, the adversary must be prevented from learning even a *single* bit of private information with certainty, so a mere reduction in bandwidth is not sufficient in our setting. Fuzz avoids this problem by converting the timing channel into a

storage channel, which in turn is handled by differential privacy.

Preventing timing channels seems hopeless in the general case. Language-based designs can eliminate them for certain types of programs (Agat, 2000), but only at the expense of severely limiting the expressiveness of the programming language. Shroff and Smith (Shroff and Smith, 2008) show how to handle more general computations but may have to abort them, which can result in garbled data and/or leak information through a storage channel. In the context of a differentially private query, however, aborting individual microqueries is safe because the impact on the overall result is known to be bounded by the sensitivity of the query. As shown in Section 3.4.4, returning default values does not open a new storage channel or increase the privacy cost of the query (though it may decrease its usefulness).

Side channels: Side channels can leak private information, e.g., through electromagnetic radiation (Gandolfi et al., 2001; Quisquater and Samyde, 2001) or power consumption (Kocher et al., 1999). Many of these channels can only be exploited if the adversary is physically close to the machine that executes the queries, which is not permitted by our threat model.

Real-time systems: Some real-time systems have provisions for handling timer overrun problems in untrusted code, such as preemption or partial admission (Wilson, Cytron, and Turner, 2009). In our scenario, it would not be sufficient to simply preempt a microquery that has overshot its timeout—we must be able to terminate it *and* clean up all of its side effects *before* the timeout expires. Another approach is inferring the worst-case execution time (Wilhelm, Engblom, Ermedahl, Holsti, Thesing, Whalley, Bernat, Ferdinand, Heckmann, Mitra, Mueller, Puaut, Puschner, Staschulat, and Stenström, 2008), which is known to be difficult even for trusted code.

3.10 Conclusion

We have demonstrated that state-of-the-art systems for differentially private data analysis are vulnerable to several different kinds of covert-channel attacks from adversarial queriers. Covert channels are particularly dangerous in this context because the leakage of even a single bit of private, un-noised information completely destroys the guarantees these systems are designed to provide. We analyzed the space of potential solutions, and we presented the design of Fuzz, which represents one specific solution from this space and relies on default values and predictable transactions. Using a proof-of-concept implementation based on Caml Light, we demonstrated that Fuzz can be retrofitted into an existing language runtime. Our evaluation shows that Fuzz is practical and expressive enough to support realistic queries. Fuzz increases query completion times compared to systems without covert-channel defenses, but the increase does not seem large enough to prevent practical applications.

While we have demonstrated that Fuzz is safe against these sophisticated attacks, it is not in general feasible to mathematically prove that a system is safe against all classes of timing attacks. Indeed, the two successful attacks against Fuzz discussed in Chapter 3.9 both worked due to our lack of protections in the final post-processing step of sampling and adding Laplace noise to the calculated answer. Even state-of-the-art first principles verifications efforts, such as Hawblitzel, Howell, Lorch, Narayan, Parno, Zhang, and Zill (2014) do not cover timing channel attacks, (although they would have ruled out the floating point attack from Mironov (2012)).

4

Differentially Private Join Queries over Distributed Databases

4.1 Introduction

In this chapter, we study the problem of answering queries about private data that is spread across multiple different databases. For instance, a medical researcher may want to study a possible correlation between travel patterns and certain types of illnesses. The necessary information exists today – e.g., in airline reservation djoin-systems and hospital records – but it is maintained by two separate companies who are prevented by law from sharing this information with each other, or with a third party. This separation prevents the processing of such queries, even if the final answer, e.g., a noised correlation coefficient, would be safe to release.

4.1.1 Motivation

Existing differential privacy query processors assume either that all the data is available in a *single* database—as in the previous chapter on Fuzz, or in McSherry (2009) and Roy et al. (2010)—or that distributed queries can be broken into several subqueries that can each be answered using only one of the databases (Dwork et al., 2006a; Rastogi and Nath, 2010; Chen et al., 2012; Götz and Nath, 2011). In practice, this is not necessarily the case. For instance, suppose a medical researcher wanted to study how a certain illness is correlated with travel to a particular region. This data may be available, e.g., in a hospital database H and an airline reservation system R, but to determine the correlation, it is necessary to *join the two databases together* – for instance, we must count the individuals who have been treated for the illness (according to H) *and* have traveled to the region (according to R).

We are not aware of any existing method or query processor that can efficiently support join queries with differential privacy guarantees. Joins cannot be broken into smaller subqueries on individual databases because, in order to match up the same persons' data in the two databases, such queries would have to ask about individual rows, which is exactly what differential privacy is designed to prevent. In principle, one could process joins using secure multi-party computation (MPC) (Yao, 1982), but MPC is only practical for small computational tasks, and differential privacy only works well for large databases. The cost of an entire join under MPC would be truly spectacular.

DJoin, the system we present in this chapter, is a solution to this problem. DJoin can support SQL-style queries across multiple databases, including common forms of joins. The key insight behind DJoin is that the distributed parts of many queries can be expressed as intersections of sets or multisets. For instance, we can rewrite the query from above to locally select all patients with the illness from H and all travelers to the relevant region from R, then intersect the resulting sets, and finally count the number of elements in the intersection. Not all SQL queries can be rewritten in this

way, but many counting queries can: conjunctions and disjunctions of equality tests directly correspond to unions and intersections of data elements. As we will show, a number of additional operations, such as inequalities and numeric comparisons, can be expressed in terms of multiset operations.

Protocols for private set operations have been studied by cryptographers for some time (Vaidya and Clifton, 2005; Kissner and Song, 2005; Freedman, Nissim, and Pinkas, 2004), but existing solutions compute exact set elements or exact cardinalities, which is not compatible with differential privacy. We present *blinded, noised private set intersection cardinality (BN-PSI-CA)*, an extension of the set-intersection protocol from Kissner and Song (2005) that supports private noising, as well as *denoise-combine-renoise (DCR)*, an operator that can add or subtract multiple noised subset cardinalities without compounding the corresponding noise terms. DCR relies on MPC to remove the noise terms on its inputs and to re-noise the output, but DCR's complexity grows with the number of parties and not with the number of elements in the sets. For the queries we tried, this step never took more than 20 seconds.

We have implemented and evaluated a prototype of DJoin. Our results show that the costs are substantial but typically feasible. For instance, the elements in a simple two-way join on databases with 32,000 rows each can be evaluated in about 1.8 hours, with 83 MB of traffic, using a single commodity workstation for each database. This is orders of magnitude faster than general MPC. DJoin's cost is too high for interactive use, but it seems practical for applications that can tolerate a certain amount of latency, such as research studies. Our algorithms are easy to parallelize, so the speed could be improved by increasing the number of cores.

To summarize, this chapter makes the following four contributions:

- two new primitives, BN-PSI-CA and DCR, for distributed private query processing (Section 4.4);
- a query planner that rewrites SQL-style queries to take advantage of those two

primitives (Section 4.5);

- the design of DJoin, an engine for distributed, differentially private queries (Section 4.6); and
- an experimental evaluation of DJoin, based on a prototype implementation (Section 4.7).

4.2 Related work on distributed private operations

Private set operations: The first protocols for private two-party set intersection and set intersection cardinality were proposed by Freedman et al. Freedman et al. (2004). Since then, a number of improvements have been proposed; for instance, Kissner and Song (2005) extended the protocols to multiple parties, and Vaidya and Clifton (2005) reduced the computational overhead. These protocols produce exact results, and are thus not directly suitable for differential privacy. There are specialized protocols for other private multi-party operations, e.g., for decision-tree learning (Pinkas, 2002), and some of these have been adapted for differential privacy, e.g., (Zhang, Li, and Lou, 2011).

Untrusted servers: Several existing systems enable clients to use an untrusted server without exposing private information to that server. In SUNDR (Li, Krohn, Mazières, and Shasha, 2004), SPORC (Feldman, Zeller, Freedman, and Felten, 2010), and Depot (Mahajan, Setty, Lee, Clement, Alvisi, Dahlin, and Walfish, 2010), the server provides storage; in CryptDB (Popa, Redfield, Zeldovich, and Balakrishnan, 2011), it implements a database and SQL-style queries. This approach is complementary to ours: DJoin's goal is to reveal *some* useful information about the data it stores, but with an upper bound on how much can be learned about a single individual.

Query Optimization: Our approach does not take advantage of existing work on database query optimizations: until and unless a particular optimization can be

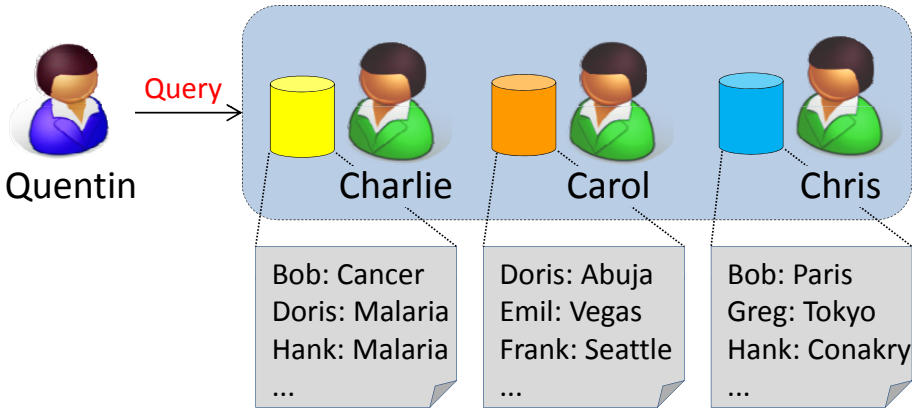


Figure 4.1: Motivating scenario. Charlie is a physician, and Carol and Chris are travel agents. Quentin would like to know the correlation between treatment for malaria and travel to high-risk areas.

proved differentially private. However, our approach has parallels to existing work in database query planning. In particular, converting our JOIN queries into BN-PSI-CA subqueries is similar to a 2-phase semijoin execution (Mackert and Lohman, 1986), except that after the two subsets are identified, rather than sending them across the network, we perform the BN-PSI-CA algorithm to privately compute the cardinality.

4.3 Background and overview

4.3.1 Motivating scenario

Figure 4.1 shows our motivating scenario. Charlie, Carol, and Chris each have a database with confidential information about individuals; for instance, Charlie could be a physician, and Carol and Chris could be travel agents. We will refer to these three as the *curators*. Quentin asks a question that combines data from each of the databases; for instance, he might want to know the correlation between treatment for malaria and travel to areas with a high risk of malaria infections. We will refer to Quentin as the *querier*.

Our goal is to build a system that can give an (at least approximate) answer to

Quentin's question while offering strong privacy guarantee to the individuals whose data is in the databases. In particular, we would like to establish an upper bound on how much *additional* information any participant of the system (queriers or curators) can learn about any individual in the database. The word 'additional' is crucial here, since the curators each have full access to their respective databases. For instance, since Charlie has treated Bob for cancer, our system cannot prevent him from learning this fact, but it can prevent him from learning whether or not Bob has recently traveled to Paris.

4.3.2 Challenge: Distribution

Answering differentially private queries over a *single* database is a well-studied problem, and several systems (McSherry, 2009; Roy et al., 2010), including Fuzz in the previous chapter, are already available for this purpose. In principle, these systems can also be used to answer queries across multiple databases, but this requires that all curators turn over their data to a single trusted entity (e.g., one of the curators), who evaluates the query on their behalf. However, there may not always be a single entity that is sufficiently trusted by all the curators, so it seems useful to have an alternative solution that does not require a trusted entity.

In some cases, distributed queries can be factored into several subqueries that can each be executed on an individual database. For instance, a group of doctors can count the number of male patients in their respective databases by counting the number of patients in *each* database separately, and then add up the (individually noised) results. This type of distributed query is supported by several existing systems (Dwork et al., 2006a; Rastogi and Nath, 2010; Chen et al., 2012; Shi, Chan, Rieffel, Chow, and Song, 2011b; Götz and Nath, 2011). However, not all queries can be factored in this way. For instance, the above approach will double-count male patients that have been treated by more than one doctor, but a union query (which would avoid this problem) cannot be expressed as a sum of counts. Similarly,

any query that involves joining several databases (such as our motivating example) cannot be expressed in this way.

Joins *could* be supported via general-purpose multi-party computation (MPC) (Yao, 1982), but the required runtimes would be gigantic: state-of-the-art MPC solutions, such as FairplayMP (Ben-David, Nisan, and Pinkas, 2008), need about 10 seconds to evaluate (very simple) functions that can be expressed with 1,024 logic gates. Since the number of gates needed for a join would be at least quadratic in the number of input rows, and since differential privacy only works well for large databases, this approach does not seem practical.

4.3.3 Approach

The key insight behind our solution is that joins are rarely used to compute full cross products of different databases; rather, they are often used to 'match up' elements from different databases. For instance, in our running example, we can first select all the individuals in R who have traveled to the region of interest, then select all the individuals in H who have been treated for the illness, and finally count the number of individuals who appear in both sets. Thus, the problem of privately answering the overall query is reduced to 1) some local operations on each database, and 2) privately computing the cardinality of the intersection of multiple sets. Not all queries can be decomposed in this way, but, as we will show in Section 4.5, there is a substantial class of queries that can.

Protocols for private multiset operations (such as intersection and union) are available (Vaidya and Clifton, 2005; Kissner and Song, 2005; Freedman et al., 2004), but they tend to compute *exact* sets or set cardinalities. If we naïvely used these algorithms, Charlie could compute the intersection of the set of the malaria patients in his database with the sets of customers in Carol's and Chris' databases who have traveled to high-risk areas, and then add noise in a collaborative fashion (Dwork et al., 2006a). This would prevent Quentin from learning anything other than the

(differentially private) output of the query — but Charlie could learn where his patents have traveled, and Carol and Chris could learn which of their customers have been treated for malaria. Hence, our first challenge is to extend these set-intersection operations to support noising between the data curators.

A second challenge arises because some queries involve multiple set operations. If Charlie simply added the two cardinalities together, the noise terms would compound, and thus (unnecessarily) degrade the quality of the overall result. To avoid this problem, we need a way to de-noise, combine, and re-noise intermediate results without compromising privacy.

4.4 Building blocks: BN-PSI-CA and DCR

Next, we describe two key building blocks that enable private processing of distributed queries. Each building block performs only one, very specific operation. In Section 4.5, we will describe how these building blocks can be used in a larger query plan to answer a variety of different queries.

4.4.1 Background: PSI-CA

Our first building block is related to a primitive called *private set-intersection cardinality (PSI-CA)*, which allows a group of k curators with multisets $S_1, \ldots, S_k$ to privately compute $|\bigcap_i S_i|$, i.e., the (exact) number of elements they have in common, but *not* the specific elements in $\bigcap_i S_i$. PSI-CA is a well-studied primitive (Freedman et al., 2004; Kissner and Song, 2005; Vaidya and Clifton, 2005), albeit not in the context of differential privacy. To explain the intuition, we describe one simple PSI-CA primitive (Freedman et al., 2004) for only two curators with simple sets in the honest-but-curious (HbC) model. The primitive uses a homomorphic encryption scheme that preserves addition and allows multiplication by a constant. Paillier's cryptosystem (Paillier, 1999) is an example of a scheme that has this property.

Suppose the two curators are C_1 and C_2 and their sets are $S_1 := \{x_1, \ldots\}$ and

$S_2 := \{y_1, \ldots\}$. C_1 defines a polynomial $P(z)$ over a finite field whose roots are his set elements x_i:

$$P(z) := (x_1 - z)(x_2 - z)\cdots = \sum_u \alpha_u z^u$$

Next, C_1 sends homomorphic encryptions of the coefficients α_u to C_2, along with the public key. For each element $y_i \in S_2$, C_2 then computes $\mathrm{Enc}(rP(y_i) + 0^+)$, i.e., she evaluates the polynomial at each of her inputs, multiplies each result by a fresh random number r, and finally adds a special string 0^+, e.g., a string of zeroes. Since the cryptosystem is homomorphic, C_2 can do this even though she does not know C_1's private key. Finally, C_2 sends a random permutation of the results back to C_1, who decrypts them and counts the occurrences of the special string 0^+, which is exactly $|S_1 \bigcap S_2|$.

At first glance, the cost of this algorithm appears to be quadratic: C_2 must compute $\mathrm{Enc}(rP(y_i) + 0^+)$ for each of her $|S_2|$ inputs, which involves computing $\mathrm{Enc}(P(y_i))$ along the way. If this is naïvely evaluated as $\mathrm{Enc}(\sum_{u=0}^{|S_1|} \alpha_u y_i^u)$, C_2 must multiply each of the $|S_1| + 1$ encrypted coefficients with an unencrypted constant (y_i^u), which requires an exponentiation each time, for a total of $O(|S_1| \cdot |S_2|)$ exponentiations. However, Freedman et al. (2004) describes several optimizations that can reduce this overhead, including an application of Horner's rule and the use of hashing to replace the single high-degree polynomial with several low-degree polynomials. This reduces the computational overhead to $O(|S_1| + |S_2| \ln\ln |S_1|)$ exponentiations.

4.4.2 BN-PSI-CA: Two-party case

The basic PSI-CA primitive is not compatible with differential privacy because C_1 learns the *exact*, un-noised size of $|S_1 \bigcap S_2|$; moreover, each curator can learn the size of the other curator's set by observing the number of encrypted coefficients, or encrypted return values, that are received from that curator. However, we can extend the primitive to avoid both problems.

First, we need to make the number of coefficients and return values independent of the set sizes. We can do this by adding some extra elements that cannot appear in either of the sets. As long as we can ensure that C_1 and C_2 are adding different elements (e.g., by setting some bit to zero on C_1 and to one on C_2), this will not affect the size of the intersection. In DJoin, we assume that a rough upper bound on the size of each curator's database is known, and we add enough elements to fill up both sets to that upper bound.

Second, we need to add some noise n to the result that is revealed to C_1. We observe that C_2 can increase the apparent size of the intersection by n if she adds n different[1] encodings of the special string 0^+. However, to guarantee ε-differential privacy, we would have to draw n from a Laplace distribution $\mathrm{Lap}(1/\varepsilon)$, and this would sometimes yield $n < 0$ – but C_2 cannot *remove* encodings of 0^+ because she does not have C_1's private key, and thus cannot tell them apart from encodings of other values. Instead, we require C_2 to draw n from $X_2 + \mathrm{Lap}(1/\varepsilon)$ and we cut n at 0 and $2 \cdot X_2$; thus, C_2 can add n encodings of 0^+ and $2 \cdot X_2 - n$ encodings of a random value to keep the overall size independent of n. (Cutting the Laplace distribution can leak a small amount of information when the extremal values are drawn, and thus changes the privacy guarantee to (ε, δ)-differential privacy (Dwork et al., 2006a); however, by increasing X_2, we can make δ arbitrarily small, at the expense of a higher overhead.) We call the resulting primitive *blinded noised PSI-CA (BN-PSI-CA)*.

Note that at the end, C_2 knows the noise term n and C_1 the noised cardinality $|S_1 \bigcap S_2| + n$. Thus, if the latter is used in further computations, we have an opportunity to remove the noise again, as long as we can ensure that neither curator learns both values. This prevents the noise terms from compounding, and it enables us to use a *very* high noise level (and thus a low value of ε) because the noise will not affect the final result.

[1] The Paillier cryptosystem can construct many different ciphertexts for the same plaintext.

4.4.3 BN-PSI-CA: Multi-party case

Since Freedman's initial work, cryptographers have considerably extended the range of private multiset operations. For instance, the protocol by Kissner and Song (2005) also supports set unions, as well as set intersections with more than two parties, and it is *compositional*: the result of a set union or set intersection can be unioned or intersected with further sets, without decrypting it first. Kissner and Song (2005) can evaluate any function on multisets that can be described by the following grammar:

$$\Upsilon ::= s \,|\, \Upsilon \cap \Upsilon \,|\, s \cup \Upsilon \,|\, \Upsilon \cup s$$

where s is a multiset that is known to some curator C_i.

The protocol from (Kissner and Song, 2005) computes $|\bigcap_{i=1,\ldots,k} S_i|$ as follows. First, the k curators use a homomorphic threshold cryptosystem to share a secret key sk amongst themselves, while the corresponding public key pk is known to all curators. Each curator C_i now encrypts a polynomial P_i whose roots are the elements of its local set S_i. The encrypted polynomials are then essentially added together, yielding a polynomial P whose roots are the elements in the intersection. Each curator C_i now evaluates P on the elements e_{ij} of his local set S_i, yielding values $v_{ij} := P(e_{ij})$; however, recall that, because sk is shared, no individual curator can decrypt the v_{ij}. The curators then securely re-randomize and shuffle (Neff, 2001) the v_{ij}, such that each curator learns all the v_{ij} but cannot tell which curator it came from. Finally, the curators jointly decrypt the v_{ij}. If there are n elements in the intersection, this yields $n \cdot k$ zeroes; hence, each curator can compute the final result by dividing the number of zeroes by k.

We can use the same blinding technique as in Section 4.4.2 to construct a multi-party version of BN-PSI-CA. After computing the v_{ij}, but before the shuffle, each curator draws a noise term n_i as above and adds $2 \cdot X_i$ extra values, n_i of which are 0^+. As above, this adds $\sum_i n_i$ to the resulting cardinality, but the noise can be removed

again via DCR, which we discuss next.

4.4.4 DCR: Adding cardinalities

BN-PSI-CA is sufficient to answer queries that require a single distributed multiset operation. However, in Section 4.5.2 we will see that some queries require multiple operations, and that the result is then a linear combination of the different cardinalities. In principle, we could designate a single curator C that collects all cardinalities and computes the overall result; however, this would a) compound all the noise terms and thus decrease the quality of the result, and b) reveal all the intermediate results to C and thus (unnecessarily) reveal some private information.

Instead, we can combine the various cardinalities using secure multi-party computation (MPC) (Yao, 1982). If we have a number of players with private inputs x_i that are each known to only one of the players, MPC allows the players to collectively compute a function $f(x_1, x_2, \ldots)$ *without* revealing the inputs to each other. Even after decades of research, MPC remains impractical for complex functions or large inputs, but modern implementations, such as (Ben-David et al., 2008), can process simple functions in a few seconds or less. Thus, while MPC may be too expensive to evaluate the entire query, we can certainly use it to combine a small number of subquery results.

For instance, suppose the query is for $|S_1 \bigcap S_2| + |S_3 \bigcap S_4|$, and that there are four curators involved: C_1 and C_3 learn the noised results R_1 and R_2 for the first and the second term, respectively, and C_2 and C_4 learn the corresponding noise terms n_1 and n_2. Then we can compute the query result under MPC as

$$q = R_1 + R_2 - (n_1 + n_2) + N$$

where each of the four curators contributes one of the private inputs R_i and n_i, and N is a new, global noise term. Next, we describe how N is computed.

		From	To
R1	Local sel.	$\sigma_{P(X)\wedge Q}(X \bowtie Y)$	$\sigma_Q(\sigma_P(X) \bowtie Y)$
R2	Disjunction	$\sigma_{P\vee Q}(X \bowtie Y)$	$\sigma_P(X \bowtie Y) \bigcup \sigma_Q(X \bowtie Y)$
R3	Split	$\lvert\sigma_{X.a=Y.b\wedge(P(X)\vee Q(Y))}(X \bowtie Y)\rvert$	$\lvert\sigma_{X.a=Y.b}(\sigma_P(X) \bowtie Y)\rvert +$ $\lvert\sigma_{X.a=Y.b}(\sigma_{\neg P}(X) \bowtie \sigma_Q(Y))\rvert$
R4	Union	$\lvert X \bigcup Y\rvert$	$\lvert X\rvert + \lvert Y\rvert - \lvert X \bigcap Y\rvert$
R5	Not equal	$\lvert\sigma_{X.a=Y.b\wedge X.c\neq Y.d}(X \bowtie Y)\rvert$	$\lvert\sigma_{X.a=Y.b}(X \bowtie Y)\rvert - \lvert\sigma_{X.a=Y.b\wedge X.c=Y.d}(X \bowtie Y)\rvert$
R6	Comparison	$\lvert\sigma_{X.a=Y.b\wedge X.c>Y.d}(X \bowtie Y)\rvert$	$\sum_{i=0..k-1} \lvert\pi_{\text{a}\,\lvert\rvert\,\text{pre(c,i)}}(\sigma_{\text{bit(c,i)}=1}(X)) \bigcap$ $\pi_{\text{b}\,\lvert\rvert\,\text{pre(d,i)}}(\sigma_{\text{bit(d,i)}=0}(Y))\rvert$
R7	Equality	$\lvert\sigma_{X.a=Y.b\wedge X.c=Y.d}(X \bowtie Y)\rvert$	$\lvert\sigma_{(X.a\lvert\rvert\text{pad}_{\lvert\rvert X.c)=(Y.b\lvert\rvert\text{pad}_{\lvert\rvert Y.d)}}(X \bowtie Y)\rvert$
R8	Join	$\lvert\sigma_{X.a=Y.b}(X \bowtie Y)\rvert$	$\lvert\pi_a(X) \bigcap \pi_b(Y)\rvert$

Table 4.1: DJoin's rewrite rules. These rules are used to transform a query (written in the language from Figure 4.2) into the intermediate query language from Figure 4.4, which can be executed natively.

4.4.5 DCR: Cooperative noising

MPC enables us to safely remove the noise that was added to the individual cardinalities by BN-PSI-CA, but we must add back a sufficient amount of noise N *as part of the MPC*, i.e., before the result is revealed. To prevent information leakage, the new noise N must be such that no individual curator can control it or predict its value.

We follow the algorithm in Dwork et al. (2006a) to generate the noise N, with some implementation modifications. Each curator chooses a random bitstring v_i uniformly at random and contributes it as an input to the MPC. The MPC computes $v := v_1 \oplus v_2 \oplus \cdots$. As long as a curator honestly chooses v_i uniformly at random and does not share this with any other party, she can be certain that no other curator can know anything else about the computed noise string v, even if every single other curator colludes. Finally, the MPC uses the fundamental transformation law of probabilities to change the distribution of v to a Laplace distribution $\text{Lap}(1/\varepsilon)$. This yields the noise term N, which is then added to the query result. We call this primitive *denoise-combine-renoise (DCR)*.

```
query     := SELECT output FROM union
             WHERE predicate
output    := NOISY COUNT(field)
union     := rows | union UNION ALL rows
rows      := join | subquery
join      := db{,db}*
subquery  := SELECT fields FROM join
             WHERE predicate
predicate := term | predicate OR term |
             predicate AND term
term      := val = val | val != val |
             val < val
val       := number | string | db.field
```

Figure 4.2: DJoin's query language.

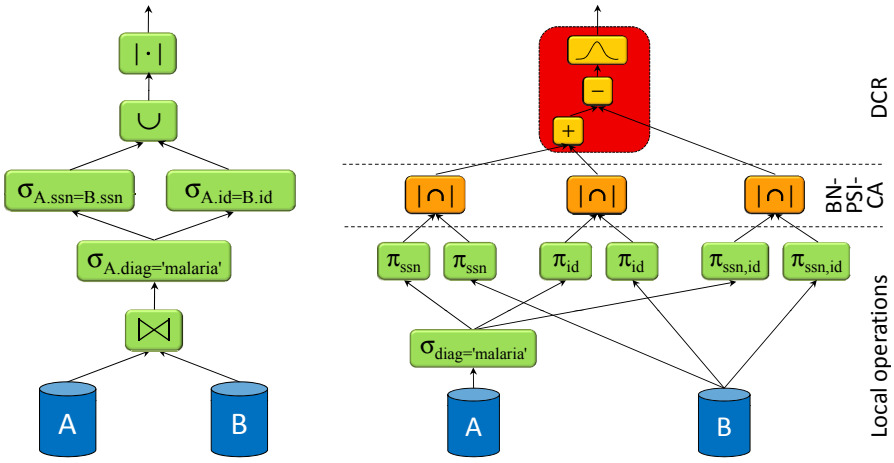
4.5 Distributed query processing

So far, we have described BN-PSI-CA, which can compute differentially private set intersection cardinalities, and DCR, which can privately combine multiple cardinalities. Next, we describe how DJoin integrates these two primitives into larger query plans that can answer SQL-style queries.

4.5.1 Query language: SPJU

For ease of presentation, we describe our approach using the simple query language in Figure 4.2, which consists of SQL-style operators for selection, projection, a cross join, and union (SPJU). This query language is obviously much simpler than SQL itself, but it is rich enough to capture many interesting distributed operations. We note that many of the missing features of SQL can easily be added back, as long as queries do not use them to access more than one database at a time.

Each query in our language can be translated into relational algebra, specifically, in a combination of selections (σ), projections (π), joins ($\bowtie$), unions ($\bigcup$), and counts



(a) Original plan (b) Rewritten for DJoin

Figure 4.3: Query example. The original plan (left) cannot be executed without compromising privacy. The rewritten plan (right) consists of three tiers: a local tier, a BN-PSI-CA tier, and a DCR tier.

$(|\cdot|)$. For instance, the query

```
SELECT COUNT(A.id) FROM A,B
WHERE (A.ssn=B.ssn OR A.id=B.id)
  AND A.diagnosis='malaria'
```

could be written (with abbreviations) as:

$$|\,\sigma_{(\text{A.ssn}=\text{B.ssn}\vee\text{A.id}=\text{B.id})\wedge\text{A.diag}=\text{"malaria"}}(A\bowtie B)\,|$$

Figure 4.3(a) shows a graphical illustration of this query.

4.5.2 Query rewriting

Most distributed queries cannot be executed natively by DJoin because they contain operators (such as $\bowtie$ or $<$) that our system cannot support. Therefore, such queries must be transformed into other queries that are semantically equivalent but contain only operators that our system *can* support, which are a) any SQL queries on a single database that produce a noisy count or a multiset; b) BN-PSI-CA; and c) DCR.

```
query    := cardex | cardex + cardex |
            cardex - cardex
cardex   := |setex ∩ setex {∩ setex}*|
setex    := π_fields(sigmaex) | sigmaex
sigmaex  := σ_local_predicate(db) | db
```

Figure 4.4: DJoin's intermediate language.

Figure 4.4 shows the language that can be supported natively. DJoin uses a number of rewrite rules to perform this transformation. The most interesting rules are shown in Table 4.1; some trivial rules, e.g., for transforming boolean predicates, have been omitted.

Local selects: We try to perform as many operations as possible locally at each database, e.g., via rule R1 for selects that involve only columns from one database.

Disjunctions: We use basic boolean transformations to move any disjunctions in the join predicates to the outermost level, where they can be replaced by set unions using rule R2, or split off using rule R3.

Unions: Rule R4 (which is basically De Morgan's law) replaces all the set unions with additions, subtractions, and set intersections.

Inequalities: Rule R5 replaces the $\neq$ operators with an equality test and a subtraction; rule R6 encodes integer comparisons as a sum of equalities. Both rules assume that there is a nearby equality test for matching rows.

Equalities: Once all non-local operations in the join predicates are conjunctions of equality tests, we can use rule R7 to reduce these to a single equality test, simply by concatenating the relevant columns in each database (with appropriate padding to separate columns).

Joins: Once a join cardinality has only one equality test left, rule R8 replaces it with an intersection cardinality.

4.5.3 Result: Three-tier query plan

If the rewriting process has completed successfully, the rewritten query should now conform to our intermediate language from Figure 4.4, which implies a three-tier structure: the first tier (`sigmaex` and `setex`) consists of local selections and projections that involve only a single database; the second tier (`cardex`) consists of set intersection cardinalities, and the third tier (`query`) consists of arithmetic operations applied to cardinalities. We refer to the rewritten query as a *query plan*. Figure 4.3(b) shows a query plan for the query from Figure 4.3(a) as an illustration.

A query plan with this three-tier structure can be executed in a privacy-preserving way. The first tier can be evaluated using classical database operations on the individual databases; the second tier can be evaluated using BN-PSI-CA (Section 4.4.2 and 4.4.3), and the third tier can be evaluated using DCR (Section 4.4.4 and 4.4.5).

4.5.4 Limitations

DJoin has only two distributed operators: BN-PSI-CA and DCR. If a query cannot be rewritten into a query plan that uses only those operators (and some purely local ones), it cannot be supported by DJoin. For instance, DJoin currently cannot process the query

```
SELECT COUNT(A.id) FROM A,B,C
WHERE ((A.x*B.y)<C.z)
```

because we know of no efficient way to rewrite the predicate into set intersections. Rewriting is generally difficult for predicates that involve computations across fields from multiple databases. The predicates DJoin *can* support include 1) predicates that use only fields from a single database, 2) equality tests between fields from different databases, and 3) conjunctions and disjunctions of such predicates. In addition, DJoin supports operators for which it has an explicit rewrite rule, such as inequalities and numeric comparisons (rules R5 and R6). We do not claim that we have found

all possible rewrite rules; if rules for additional operators are discovered, DJoin could be extended to support them as well.

DJoin is currently limited to counting queries: it does not support sum queries, or queries with non-numeric results. Differential privacy can in principle support such queries, e.g., via the exponential mechanism (McSherry and Talwar, 2007), but we have not yet found a way to express them in terms of set intersections.

4.6 DJoin design

In this section, we present the design of DJoin, our system for processing distributed differentially-private queries using the mechanisms explained so far.

4.6.1 Assumptions

Our design is based on the following assumptions:

1. All queriers know the schema and a rough upper bound on the total size of each curator's database.
2. The curators are "honest but curious", i.e., they will learn whatever information they can, but they will not deviate from the protocol.
3. Each curator has a "privacy budget" that represents to amount of private information he or she is willing to release through queries.
4. The curators can authenticate each querier.

Assumption 1 is necessary to make BN-PSI-CA and query planning work. Assumption 2 is not inherent (PSI-CA can work in an adversarial model (Kissner and Song, 2005)) but helps with efficiency and does not seem unreasonable in practice. Assumption 3 is common for differentially private query processors (McSherry, 2009; Roy et al., 2010), and as in Fuzz in Chapter 3. and assumption 4 can be satisfied, e.g., using cryptographic signatures.

4.6.2 Overview and roadmap

DJoin consists of a number of *servers*, which run on the curators' machines, as well as at least one *client*, which runs the querier's machine and communicates with the servers to execute queries. Each server has a privacy budget (Section 4.6.3) and a local database with a schema (Section 4.6.4) that is known to all clients and servers.

Users can interact with DJoin by issuing a query q and a requested accuracy level v to their local client. (v is the parameter of the Laplace distribution from which DCR will draw the final noise term.) The user's client attempts to rewrite the query according to the rules from Section 4.5.2. If this succeeds, the result is a different query q' that is equivalent to q but can be executed entirely with local queries, BN-PSI-CA, and DCR. The client then submits the query to the servers, and each server performs an analysis (Section 4.6.5) to determine the sensitivity $S(q, db_i)$ of the query q in that server's local data db_i. In combination with the accuracy level v, the sensitivity yields the privacy cost ε_i that this server will incur for answering the query.

Next, the client then uses a distributed commit protocol (Section 4.6.6) to assign an identifier to the query and to ensure that all the servers agree which query is being executed. Once the query is committed, the servers execute the query in three stages (Section 4.6.7): first, each server completes any subqueries that involve only its local database; next, the servers jointly complete each of the BN-PSI-CA operations; and finally, the servers execute DCR to combine and re-noise their results. The overall result is then revealed to the client.

4.6.3 Privacy budget

Each server maintains three pieces of local information: A local database, a privacy budget, and a table of pending queries, which is initially empty.

The *privacy budget* is essentially an upper bound on the amount of private information about any individual that the curator owning the server is willing to release

through answering queries. It is well known (Dwork, 2006) that, if q_1 and q_2 are two queries that are ε_1- and ε_2-differentially private, respectively, the sequential composition of both is $(\varepsilon_1 + \varepsilon_2)$-differentially private. Because of this, servers can simply deduct each query's "privacy cost" from the budget separately, without having to remember previous queries. A similar construction is used in other differentially private query processors, including PINQ (McSherry, 2009), Airavat (Roy et al., 2010), and the Fuzz system we described in Chapter 3. In the appendix to this chapter, we briefly sketch a possible approach to choosing the privacy budget.

Recall from Section 4.4 that DJoin must charge the privacy budget both for intermediate results from BN-PSI-CA operations and for the final result that is revealed by DCR. To avoid confusion, we use the symbol ε_p to denote the cost of a BN-PSI-CA operation and ε_r to denote the cost of the final result. The total cost of a query with several BN-PSI-CAs is thus $\varepsilon_r + \sum_j \varepsilon_{p,j}$.

4.6.4 Schemata and multiplicities

The local database is a relational database that can be maintained in a classical, non-distributed DBMS, e.g., mySQL. For simplicity, we will assume that the data from each individual user is collected in a single row of the database; if this is not the case already, a normalization step (e.g., a `GROUP BY`) must be performed first. The database schema may assign an arbitrary type $\tau(c)$ to each column c; however, to make our sensitivity analysis work, we additionally allow each column to be annotated with a *multiplicity* $m(c)$ that indicates how often any individual value can appear in that column (for instance, $m(c) = 1$ indicates a column of unique keys). If no annotation is present, DJoin assumes $m(c) = \infty$.

Multiplicities are important to determine an upper bound on sensitivity of a query. Recall from Chapter 2 that the sensitivity $S(q, db_i)$ of a counting query q in a database db_i is the largest number of rows that a change to a single row in D can cause to be added or removed from the result of q. For instance, consider the query

```
SELECT COUNT(A.x) FROM A,B
WHERE A.x=B.y
```

If the multiplicities are $m(A.x) = 3$ and $m(B.y) = 5$, then a change to a single row in A can add at most five rows to the result – hence, whatever the new value of $A.x$ is, we know that B can contain at most five rows whose y-column matches that value. (The argument for disappearing rows is analogous.) Conversely, the query's sensitivity in B is three because at most three rows in A can have the value $B.y$ in column x. Note that processing such queries as intersections requires an extra encoding step; see the appendix for details.

Clearly, the use of a column with unbounded multiplicity can cause the sensitivity to become unbounded as well. However, it is safe to use such columns in conjunction with others; for instance, the query

```
SELECT COUNT(A.x) FROM A,B
WHERE A.x=B.y AND A.p=B.q
```

has sensitivity 5 in A even if $m(A.p) = m(B.q) = \infty$.

It may seem tempting to let DJoin choose the multiplicity itself, based on how often elements *actually* occur in the database. However, this would create a side channel: queriers could learn private facts about the database by observing, e.g., how much is deducted from the privacy budget after running certain queries. To avoid this problem, DJoin follows the approach from Fuzz and determines the multiplicity statically, without looking at the data.

4.6.5 Sensitivity analysis

We now describe how to infer the sensitivity of more complex queries, and specifically on the question how much the number of rows output by a query $\sigma_{\text{pred}}(db_1 \bowtie \cdots \bowtie db_k)$ can change if a single row in one of the db_i is changed.

To explain the intuition behind our analysis, we begin with a few simple examples:

1. $A \bowtie B \bowtie C$

2. $\sigma_{\text{A.x=B.y}}(A \bowtie B \bowtie C)$

3. $\sigma_{\text{A.x=B.y}\wedge\ \text{B.y=C.z}}(A \bowtie B \bowtie C)$

4. $\sigma_{\text{A.x=B.y}\wedge\ \text{A.p=B.q}}(A \bowtie B \bowtie C)$

5. $\sigma_{\text{A.x=B.y}\wedge\ \text{B.y=C.z}\wedge\ \text{A.x=C.q}}(A \bowtie B \bowtie C)$

Since query (1) has no predicates, its sensitivity in A is simply $|B| \cdot |C|$. The addition of the constraint $A.x = B.y$ changes the sensitivity to $m(B.y) \cdot |C|$, since each row in A can now join with at most $m(B.y)$ rows in B; similarly, adding $B.y = C.z$ in query (3) reduces the sensitivity to $m(B.y) \cdot m(C.z)$. When there is a conjunction of multiple constraints between the same databases, the most selective one 'wins'; hence, the sensitivity of query (4) is $\min(m(B.y), m(B.q)) \cdot |C|$. When there are multiple 'join paths', the most restrictive one wins. For instance, in query (5), the third constraint reduces the sensitivity in A only if $m(C.q) < m(B.y) \cdot m(C.z)$; otherwise, the sensitivity is the same as for query (3).

To solve this problem in the general case, we adapt a classical algorithm from the database literature (Krishnamurthy, Boral, and Zaniolo, 1986) that was originally intended for query optimization in the presence of joins. This algorithm builds a *join graph* G that contains a vertex for each database that participates in the join, and a directed edge between each pair (db_1, db_2) of vertices that is initially annotated with $|db_2|$, the size of the database db_2. We then consider each of the predicates in turn and update the edges. Specifically, for each predicate $db_i.f_1 = db_j.f_2$ with $db_i \neq db_j$, we change the annotation $w_{i,j}$ on the edge (db_i, db_j) to $\min(w_{i,j}, m(db_j.f_2))$ and, correspondingly, the annotation $w_{j,i}$ on (db_j, db_i) to $\min(w_{j,i}, m(db_i.f_1))$. Then we can obtain an upper bound on the sensitivity $S(q, db_i)$ of q in some database db_i by finding the min-cost spanning tree that is rooted at db_i, using the product of the edge annotations as the cost function.

If the predicate contains disjunctions, we can rewrite it into DNF and then add up the sensitivity bounds. This is sound because $\sigma_{p \vee q}(X) = \sigma_p(X) \bigcup \sigma_q(X)$. If a row is removed from X and the sensitivities of p and q are s_p and s_q, this can change the cardinalities of the two sets by at most s_p and s_q, and thus the cardinality of the union by at most $s_p + s_q$. The same approach also works for unions of subqueries.

4.6.6 Distributed commit

Next, we describe how the client submits the query to the servers. It is important to ensure that the servers agree on which query they are executing; without this, a malicious client could trick a server into believing that it is executing a low-sensitivity query, and thus cause an insufficient amount of noise to be added to the result. Note that there is no need to agree on an ordering because all queries are read-only.

When the client accepts a query q with requested noise level ν from the user, it first calculates the sensitivity of q and the corresponding ε; then it tries to rewrite q into an equivalent query q' that uses only the language from Figure 4.4. If this succeeds, the client chooses a random identifier I and sends a signed prepare(I, q, q', ν) message to each server. What follows is essentially a variant of the classical two-phase commit protocol.

Upon receiving the prepare message, the server at each C_i verifies that q can be rewritten into q', and that it does not already have a pending query with identifier I. If either test fails, the server responds with a nak immediately. Otherwise, C_i's server calculates its privacy cost $\varepsilon_i := \varepsilon_{r,i} + \sum_j \varepsilon_{p,ij}$ that it would incur by executing its part of q'. This cost consists of the base cost $\varepsilon_{r,i} := S(q, db_i)/\nu$, which depends on the query's sensitivity in C_i's local data, and an additional charge $\varepsilon_{p,ij}$ for each PSI-CA operation that C_i must participate in to execute q'. If C_i's privacy budget can cover ε_i, its server deducts ε_i from the budget, adds $(I, q', \nu, \varepsilon_i)$ to its pending table, and sends a signed response ack(I, q, q', ν) back to the client. Otherwise, the server responds with a nak. This might occur, for instance, if the sensitivity of q is too high or the

requested noise level ν is too low.

If the client receives at least one NAK, it sends a signed ABORT(I) message to each server that has responded with an ACK, which causes the reserved parts of the privacy budget to be released. Otherwise the client combines the received ACK messages to form a certificate Γ, and it sends COMMIT(I,Γ) to the servers. The servers verify that all required ACKs are present; if so, they begin executing the query.

4.6.7 Query execution

Each query is executed in three stages. First, upon receiving the COMMIT message, the server at each C_i computes the parts of the query that require only data from its local database db_i. For some queries, this will yield part of the result directly (e.g., in $|\sigma_{x=0}(A) \bigcup \sigma_{x=1}(B)|$), but more typically the first stage will produce a number of sets on each server that will be used as inputs in the second stage.

The second stage consists of a number of BN-PSI-CA instances. Since all servers agree on the query q', each server can independently determine which BN-PSI-CA instances it should be involved in, and what role in the protocol it should play in each instance. Ties are broken deterministically, and the instances are numbered in order to distinguish different instances that involve the same set of servers. At the end of the second stage, each server has learned a number of noised results and/or noise terms, which are used as inputs to the third stage.

The third stage consists of an invocation of DCR, which de-noises the results from the second stage, combines them as required by q', and then re-noises the combined result using the protocol from Section 4.4.4. Recall that the re-noising requires an additional input from each server that must be chosen uniformly at random. At the end of the third stage, each server learns the result of the multi-party computation and forwards it back to the client, which displays it to the user.

4.7 Evaluation

In this section, we report results from an experimental evaluation of DJoin. Our goal is to show that 1) DJoin is powerful enough to support useful queries; and that 2) DJoin's communication and computation overheads are low enough to be practical.

4.7.1 Prototype implementation

We have built a prototype implementation of DJoin for our experiments. Our prototype uses mySQL to store each curator's data and to execute the purely local parts of each query, and it relies on FairplayMP (Ben-David et al., 2008) to execute the secure multi-party computation. We implemented the two-party BN-PSI-CA primitive from Section 4.4.2, based on the `thep` library (THEP) for the Paillier cryptosystem. Our implementation includes the optimizations from (Freedman et al., 2004) that were already briefly described in Section 4.4.1, including the use of bucket hashing to replace the single high-degree polynomial P with a number of lower-degree polynomials. This reduces BN-PSI-CA's $O(|S_1| \cdot |S_2|)$ time complexity to $O(|S_1| + |S_2| \ln \ln |S_1|)$ and makes it highly parallelizable, with synchronization required only for the few elements that hash to the same bucket. Our prototype also supports multi-party BN-PSI-CA based on the protocol from Kissner and Song (2005) and the UTD Paillier Threshold Encryption Toolbox (Garrity and Kantarcioglu, 2012), but we do not include multi-party results here due to lack of space.

We also built a query planner that implements the rewrite rules from Section 4.5.2, as well as a backend for FairplayMP that outputs code for DCR (Section 4.4.5). To our knowledge, DCR is the first implementation of the shared noise generation algorithm described in Dwork et al. (2006a). Altogether, our prototype consists of 3,560 lines of Java code for the runtime engine, 249 lines of code in FairplayMP's custom language for the DCR primitive, and 6,776 lines of C++ code for the query planner.

	Query	#PSI-CA
Q1	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.y` $\lvert(\pi_x(A) \bigcap \pi_y(B))\rvert$	1
Q2	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.x AND (A.y!=B.y)` $\lvert(\pi_x(A) \bigcap \pi_x(B))\rvert - \lvert(\pi_{x,y}(A) \bigcap \pi_{x,y}(B))\rvert$	2
Q3	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.y AND (A.z="x" OR B.p="y")` $\lvert(\pi_x(A) \bigcap \pi_y(\sigma_{p=\text{"}y\text{"}}(B)))\rvert + \lvert(\pi_x(\sigma_{z=\text{"}x\text{"}}(A)) \bigcap \pi_y(\sigma_{p\neq\text{"}y\text{"}}(B)))\rvert$	2
Q4	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.x OR A.y=B.y` $\lvert(\pi_x(A) \bigcap \pi_x(B))\rvert + \lvert(\pi_y(A) \bigcap \pi_y(B))\rvert - \lvert(\pi_{x,y}(A) \bigcap \pi_{x,y}(B))\rvert$	3
Q5	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x LIKE "%xyz%" AND A.w=B.w AND (B.y+B.z>10) AND (A.y>B.y)` $\sum_{i=0..7} \lvert(\pi_{\text{w,(y}>>\text{i+1)}}(\sigma_{\text{(x like '\%xyz\%')}\wedge\text{(y\&2}^i\text{= 1)}}(A)) \bigcap \pi_{\text{w,(y}>>\text{i+1)}}(\sigma_{\text{((y+z)}>10)\wedge\text{(y\&2}^i\text{=0)}}(B)))\rvert$	8

Table 4.2: Example queries and the corresponding query plans. The number of BN-PSI-CA operations, which is a rough measure for the complexity of the query, is shown on the right.

4.7.2 Experimental setup

For our experiments, we used five Dell PowerEdge R410 machines, each with a Xeon E5530 2.4 GHz CPU, 12 GB of memory, and four 250 GB SATA disks. The machines were connected by Gbit Ethernet. Following the recommendations in Bethencourt, Song, and Waters (2006), we used 1,024-bit keys for the Paillier cryptosystem. We chose $\varepsilon_r = 0.0212$ to ensure that the noise for a query with sensitivity $s = 1$ is within ± 100 with probability 95%; we set $\varepsilon_p = 1/8 \cdot \varepsilon_r$, and we chose $\delta = 1/N = 6.67 \cdot 10^{-5}$.

Our experiments use synthetic data rather than 'real' confidential data because our cryptographic primitives operate on hashes of the data anyway, so the actual content has no influence on the overall performance. Therefore, we generated synthetic databases. Each database had $N = 15,000$ rows.

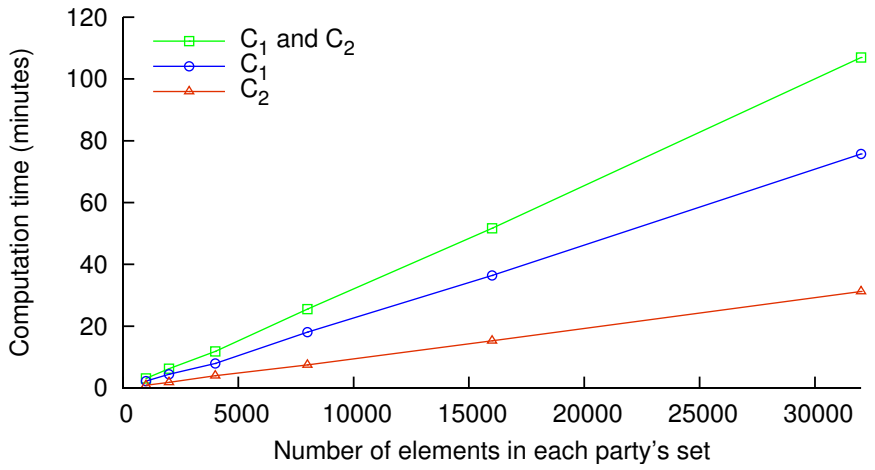


Figure 4.5: Computation time for PSI-CA. The time is approximately linear in the number of set elements.

4.7.3 Microbenchmarks: BN-PSI-CA

First, we quantified the cost of our two main cryptographic primitives. To measure the cost of BN-PSI-CA, we generated two random sets with N elements each, and we ran two-party BN-PSI-CA on them, varying N between 1,000 and 32,000 elements. We measured the computation time on each party and the amount of traffic that was exchanged between the two parties.

Figure 4.5 shows the time taken by the servers at C_1 and C_2, respectively, to execute BN-PSI-CA using a single core. The time increases almost linearly with the size of the sets; recall from Section 4.7.1 that the optimizations we applied reduce the computational overhead to $O(|S_1| + |S_2| \ln \ln |S_1|)$. Note that the two servers cannot run in parallel; the total runtime is the sum of the two servers' runtimes. Most of the computation is performed by C_1: 49% of the total time was spent constructing the polynomials at C_1; 29% of the time was spent evaluating the polynomials at C_2; and the remaining 21% were spent decrypting the resulting evaluations at C_1.

Figure 4.6 shows the total amount of traffic sent by C_1 and C_2. The traffic is roughly proportional to the set sizes. For large sets, approximately 70% of the traffic consists of polynomials sent from C_1 to C_2, and the remaining 30% consists of

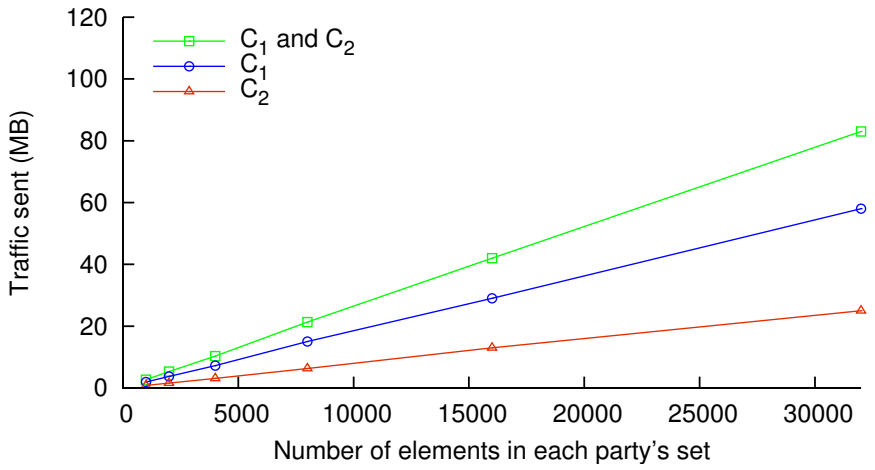


Figure 4.6: Network traffic sent by the two parties in a BN-PSI-CA run.

evaluation results sent back to C_1 for decryption.

To quantify BN-PSI-CA's scalability in the number of cores, we performed a 15,000-element intersection with one, two, and four cores. (This was done on a different machine with a 2.67 GHz Intel X3450 CPU, since our E5530s have only two cores.) The additional cores resulted in speedups of 1.99 and 3.98, respectively. This is expected because BN-PSI-CA is trivially scalable: encryptions, polynomial construction, evaluations, and decryptions can all proceed in parallel on multiple cores, or even multiple machines. Thus, DJoin should be able to handle databases much larger than 32,000 elements, as long as the computation can be spread over a sufficient number of machines.

4.7.4 Microbenchmarks: DCR

Next, we quantified the cost of the DCR operator. Recall from Section 4.4.4 that DCR internally consists of two stages: first, the inputs (cardinalities and inverted noise terms) from the various servers are added together, and then a new noise term is drawn from a Laplace distribution and added to the result. To separate the two stages, we measured the time to execute DCR twice, with and without the second stage, and we varied the number of parties from two to four.

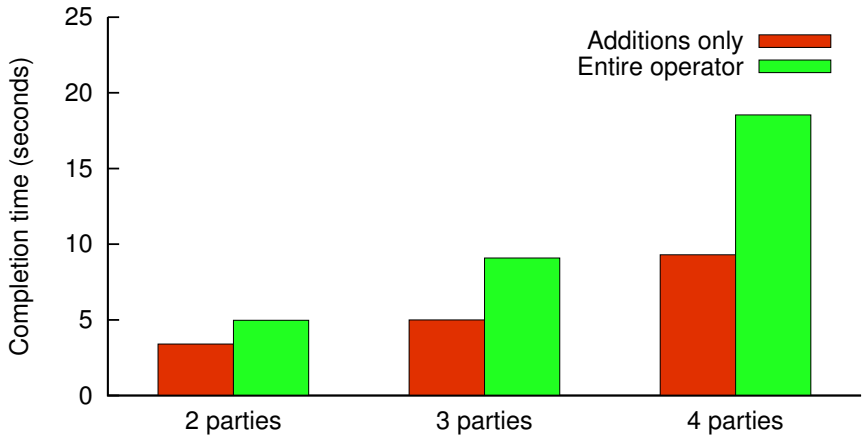


Figure 4.7: Computation time for DCR with and without the renoising step.

Figure 4.7 shows our results. The times grow superlinearly with the number of parties ((Ben-David et al., 2008) reports a quadratic dependency) but are all below 20 seconds. Although MPC is generally expensive, DJoin performs most of its work using a specialized primitive (BN-PSI-CA), so the functionality that remains for DCR to perform is fairly simple. Note that neither the size nor the number of sets affect DCR's runtime because each server inputs just a single number: the sum of all the cardinalities and noise terms it has computed.

4.7.5 Example queries

To demonstrate that DJoin can execute nontrivial and potentially useful queries, we chose five example queries, which are shown in Table 4.2 along with the query plan they are rewritten into. Each query illustrates a different aspect of DJoin's capabilities:

- **Q1** is an example of a basic join between two databases, which is transformed into a PSI-CA using rule R8.
- **Q2** adds an inequality, which is rewritten as a difference between two intersections via rule R5.

- **Q3** contains a disjunction with two local predicates, which can be split using rule R3.
- **Q4** contains another disjunction, but with remote predicates; this is rewritten via rule R2.
- **Q5** contains an equality and a numeric comparison between columns in different databases, which can be split via rule R6, as well as several other predicates that can be evaluated locally.

For Q5, the y column in both databases contained numbers between 0 and 255. The table also shows the number of BN-PSI-CA operations in each query plan, which (in conjunction with the set sizes) is a rough measure of the effort it takes to evaluate it. The more complex a query is, the more BN-PSI-CAs it requires. Q1 is the least complex query because it translates straight into a BN-PSI-CA; Q5 is the most complex one because the inequality requires one intersection per bit.

4.7.6 Query execution cost

To quantify the end-to-end cost of DJoin, we ran each of our five example queries over a synthetic dataset of 15,000 rows per database, and we measured the completion time and the overall amount of network traffic that was sent.

Figures 4.8 and 4.9 show our results. The simplest query (Q1) took 58 minutes, and the most complex query (Q5) took 448 minutes, or slightly less than seven and a half hours; the traffic was between 42.7 MB and 340 MB. Both metrics should scale roughly linearly with the size of the sets and the number of set intersections in the query, and a comparison with our microbenchmarks from Section 4.7.3 confirms this.

The completion times are much higher than the completion times one would expect from a traditional DBMS, but recall that DJoin is not meant for interactive use, but rather for occasional analysis tasks or research studies. For those purposes,

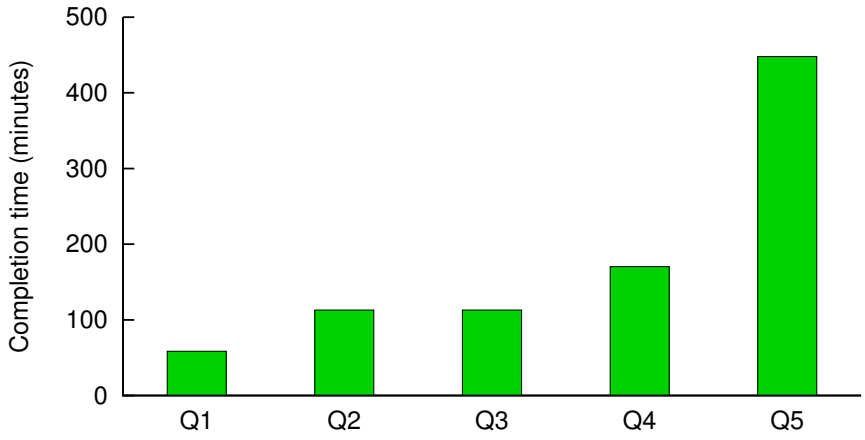


Figure 4.8: Total query execution time for each of the example queries from Table 4.2.

an hour or two should be acceptable. Also, recall that the best previously known method for executing such queries is general MPC, which is impractical at this scale.

To illustrate how much DJoin improves performance over straightforward MPC, we implemented our simplest query (Q1) directly in FairplayMP. A version for two databases of just eight (!) rows had 9,700 gates and took 40 seconds to run; we were unable to test larger databases because this produced crashes in FairplayMP. The runtimes we observed increased quadratically with the number of rows, which suggests that this approach is not realistic for the database sizes we consider. While there are other MPC frameworks that are significantly faster than FairplayMP, for two reasons, they are unlikely to be viable for executing join queries: first, the quadratic asymptotic dependence is inherent in any naive join execution, and second, much of the computational advances have been in MPC frameworks that take advantage of arithmetic circuits to optimize execution, which join queries do not benefit from as they involve large sequences of boolean choices.

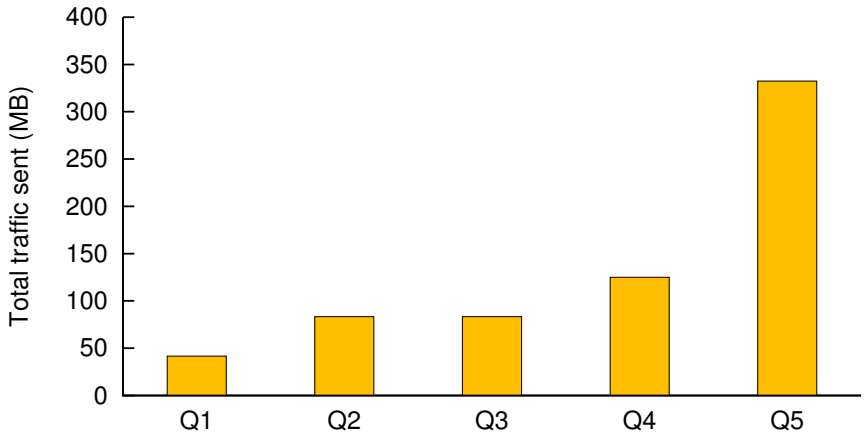


Figure 4.9: Total network traffic for each of the example queries from Table 4.2.

4.8 Conclusion

In this chapter, we have introduced two new primitives, BN-PSI-CA and DCR, that can be used to answer queries over distributed databases with differential privacy guarantees, and we have presented a system called DJoin that can execute SQL-style queries using these two primitives. Unlike prior solutions, DJoin is not restricted to horizontally partitioned databases; it supports queries that join databases from different curators together. The key insight behind DJoin is that many distributed join queries can be rewritten in terms of operations on multisets. Not all SQL queries can be transformed in this way, but many can, including counting queries with conjunctions and disjunctions of equality tests, as well as certain inequalities.

In order to provide provable privacy guarantees, DJoin does not take advantage of decades of work in database performance optimization, such as with prefetching, caching, batching, and query rewriting. This is necessarily the case, until each optimization is proven private. Without proof, we might inadvertently create a side channel, since many optimizations depend upon the actual underlying private data.

DJoin is not fast enough for interactive use, but, to the best of our knowledge, the only known alternative for distributed differentially private join queries is secure

multi-party computation, which is orders of magnitude slower. Also, most of the computational cost is due to BN-PSI-CA, which is trivially scalable and can thus benefit from additional cores.

5

Verifiable Differential Privacy

5.1 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. As we have seen in this dissertation, protecting the *privacy* of the subjects who contributed the data is clearly important, but doing so effectively is highly nontrivial. 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 (Herndon, Ash, and Pollin, 2013)), and there have been cases

where researchers appear to have fabricated or manipulated data to support their favorite hypothesis (Blake, Watt, and Winnett, March 3, 2011; Deer, February 8, 2009; Interlandi, October 22, 2006); 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.

While differential privacy 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 chapter 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 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 (Reed and Pierce, 2010; Gaboardi, Haeberlen, Hsu, Narayan, and Pierce, 2013), which uses a linear type system to certify queries as differentially private, as well as on the Pantry sys-

tem (Braun, Feldman, Ren, Setty, Blumberg, and Walfish, 2013) 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:

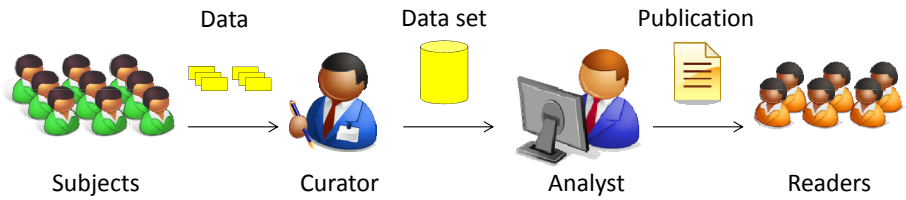


Figure 5.1: The Verifiable Differential Privacy Scenario.

- VFuzz, a query language for computations with certifiable privacy and verifiable integrity (Section 5.4);
- The design of VerDP, a system for verifiable, differentially private data analysis (Section 5.5); and
- A prototype implementation of VerDP and its experimental evaluation (Sections 5.6 and 5.7).

5.2 Overview

Figure 5.1 illustrates the scenario we address in this chapter. 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 (IPUMS), the iDASH biomedical data repository (iDASH), or the ICPSR data deposit archive (ICPSR). 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.

5.2.1 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 de-anonymized 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.

5.2.2 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 (Narayanan and Shmatikov, 2008; Barbaro and Zeller, 2006; Blake et al., March 3, 2011; Deer, February 8, 2009; Interlandi, October 22, 2006); 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 de-anonymization 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.

5.2.3 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 (Barbaro and Zeller, 2006; Bell and Koren, 2007), 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 (Narayanan and Shmatikov, 2008). To illustrate this point, and to motivate the need for solid formal foundations, we discuss a few strawman solutions below.

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 (Fournet, Kohlweiss, Danezis, and Luo, 2013), 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., (Narayanan and Shmatikov, 2008).

Naïve combination of DP and ZKP: A third approach would be to modify an

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

Figure 5.2: A very simple program for which a naïvely constructed ZKP circuit would leak confidential information.

existing differentially private data analysis tool, such as PINQ (McSherry, 2009) or Airavat (Roy et al., 2010), to dynamically output a circuit instead of executing a query directly, and to then feed that circuit to a verifiable computation system like Pinocchio (Parno, Gentry, Howell, and Raykova, 2013). Performance would obviously be a challenge, since a naïve translation could produce a large, unwieldy circuit (see Section 5.5.4); 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 5.2 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.

5.3 Background

5.3.1 Proof-based verifiable computation

VerDP builds on recent systems for *proof-based verifiable computation* (Braun et al., 2013; Parno et al., 2013). These systems make it possible for a *prover* $\mathscr{P}$ to run a program Ψ on inputs x and to then provide a *verifier* $\mathscr{V}$ 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 (Braun et al., 2013) system for verifiable computation, configured with the Pinocchio (Parno et al., 2013) 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. 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 (Reed and Pierce, 2010) — 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 chapter, 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.

5.3.2 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 $\mathscr{P}$ to read and write arbitrarily-sized data blocks from a block store. To prevent $\mathscr{P}$ 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 $\mathscr{P}$ 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 $\mathscr{V}$. One possibility is to include the database's root hash in the input x that $\mathscr{V}$ 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 con-

tains a cryptographic commitment to the database's root hash supplied by the curator (Braun et al., 2013, §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.

5.3.3 Performance of verifiable computation

Despite recent improvements, verifiable computation systems impose significant overhead on $\mathscr{P}$, often by a factor of up to 10^5 (Braun et al., 2013). 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 5.5.2). Note that commitments are relatively costly in Pantry, and so VerDP uses them sparingly.

5.4 The VFuzz language

VerDP's query language is based on the Fuzz language for differentially private data analysis (Reed and Pierce, 2010; Haeberlen et al., 2011; Gaboardi et al., 2013) 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.

5.4.1 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 func-

tions 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 (Reed and Pierce, 2010). It distinguishes between ordinary functions $\texttt{f} : \tau \to \sigma$ that can map arbitrary elements of type τ to arbitrary elements of type σ, and functions $\texttt{f} : \tau \multimap_k \sigma$ 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 and Pierce (2010) 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 5.3 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*→*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

[1] Fuzz uses the type *bag* to refer to multisets. A τbag 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 5.3: A simple program written in Fuzz that counts the individuals over 40 in a database of individuals' ages.

`sample`, and returned as the final output.

5.4.2 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 pro-

grams we are aware of can be converted easily (for more details, see Section 5.7.1). 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 this chapter.

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 Chapter 3: the data dependencies that cause the circuit size to change are the same that also affect query execution time, and vice versa. Fuzz 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.

5.4.3 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*.

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 (Dwork et al., 2006b), *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.

5.5 The VerDP system

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

5.5.1 VerDP's workflow

Figure 5.4 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

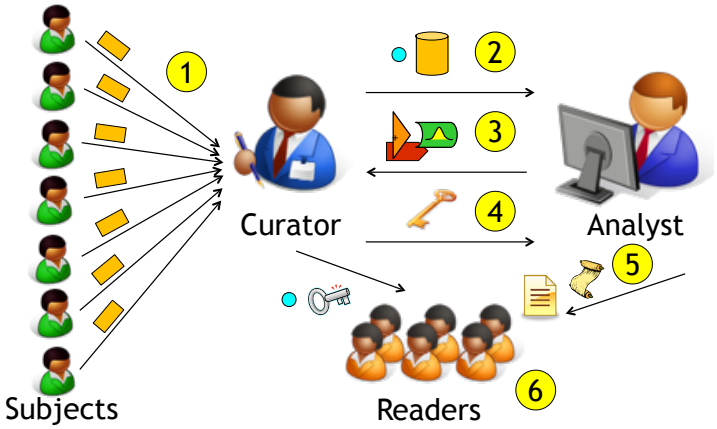


Figure 5.4: Workflow in VerDP. The numbers refer to the steps in Section 5.5.1.

db, and then publishes a commitment Comm(db) to it (Section 5.5.3). 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 5.5.2). 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

[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.

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 5.5.4). 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 5.5.5). 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.

5.5.2 The structure of VFuzz programs

VerDP's design is guided by both the properties of the VFuzz language (Section 5.4) and the performance characteristics of verifiable computation (Section 5.3.3). 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 5.3, 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

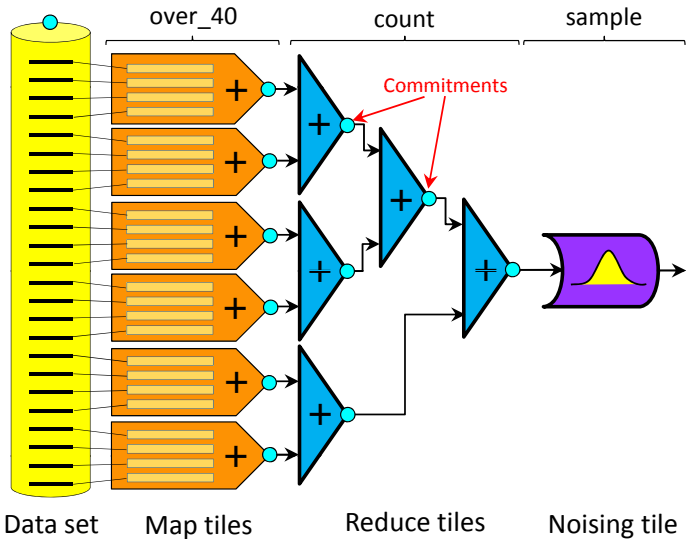


Figure 5.5: The MapReduce-like structure of VFuzz programs. The small circles represent commitments, and the labels at the top show the parts of the `over40` program from Figure 5.3 that correspond to each phase.

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 5.5. 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 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

[3]Pantry's MapReduce implementation exploits similar parallelism (Braun et al., 2013, §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.

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 5.5). 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.

5.5.3 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 5.3.2). 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 5.3.2), 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 5.5.4). 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.

5.5.4 Phases of VerDP computations

Recall from Section 5.4.3 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 5.5. 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 5.7), 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 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 $\frac{s}{\varepsilon} \cdot \ln(\bar{r})$, 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 $Lap(s/\varepsilon)$ 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 (2012), 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 (Hawblitzel et al., 2014) or the generation circuit from (Dwork et al., 2006a).

5.5.5 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 5.2.1.

5.5.6 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 (McSherry, 2009), 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 (Gaboardi et al., 2013) 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. Gaboardi et al. (2013) 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 (Barthe, Köpf, Olmedo, and Zanella-Béguelin, 2013) to complete the proof manually. However, these approaches are beyond the scope of this chapter.

5.6 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; 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 5.4.

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.

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.

Query	Type	LoC mod.	Samples	From
over-40	Counting	2 / 45	1	(Dwork et al., 2006b)
weblog	Histogram	2 / 45	2	(Dwork et al., 2006b)
census	Aggregation	9 / 50	4	(Chawla et al., 2005)
kmeans	Clustering	46 / 148	6	(Blum et al., 2005)

Table 5.1: Queries we used for our experiments (based on Fuzz 3), lines of code modified, and the inspirations.

5.7 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?

5.7.1 Queries

We used four different queries for our experiments. The first three are the queries that were used to evaluated Fuzz 3: **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 (Dwork et al., 2006b; Chawla et al., 2005; Blum et al., 2005), and each represents a different type of computation (histogram, aggregation, and clustering). We also included our running example from Figure 5.3 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 5.1 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 5.1 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.

5.7.2 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.

5.7.3 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.

5.7.4 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 5.4.3 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". 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

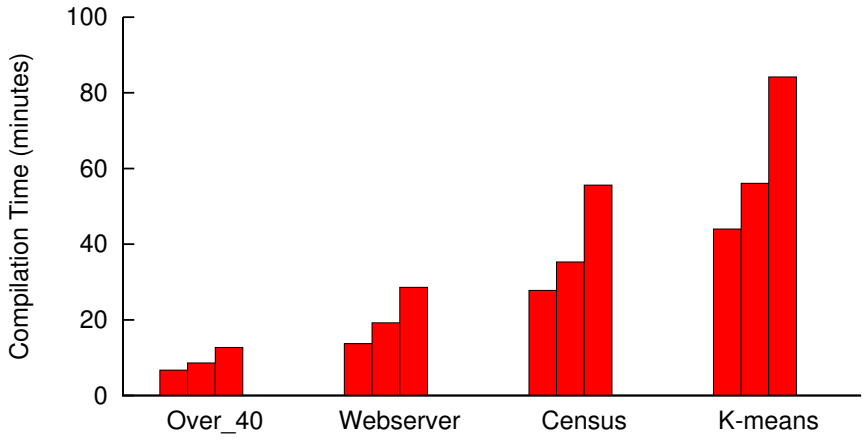


Figure 5.6: Compilation time for map tiles as a function of tile size, for (512, 1024, 2048) rows per tile.

and VKs; our results are shown in Figure 5.6. 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 5.7 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.

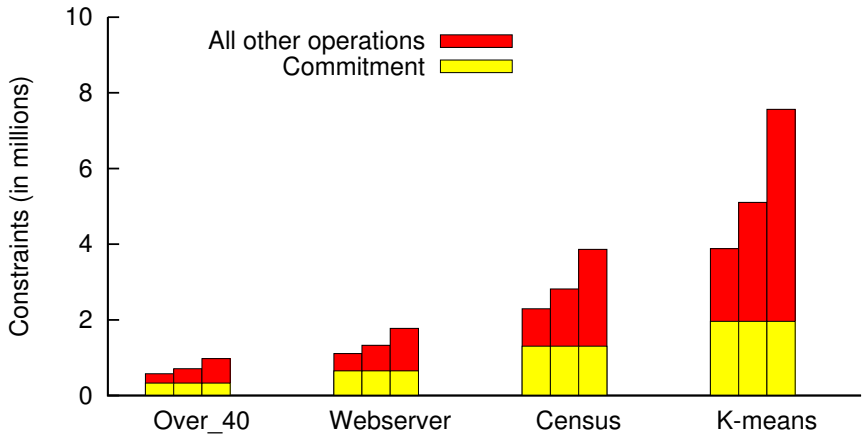


Figure 5.7: Constraints generated for map tiles with (512, 1024, 2048) rows per tile.

5.7.5 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 5.8. 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 5.5.2 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

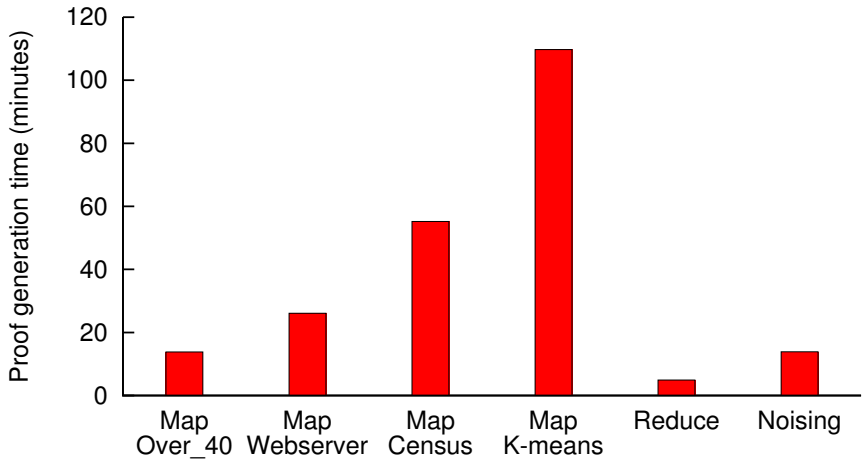


Figure 5.8: Time to generate proofs for map tiles of width 2,048, as well as the reduce and noising tiles.

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 5.9 shows our estimates; 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 5.10 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

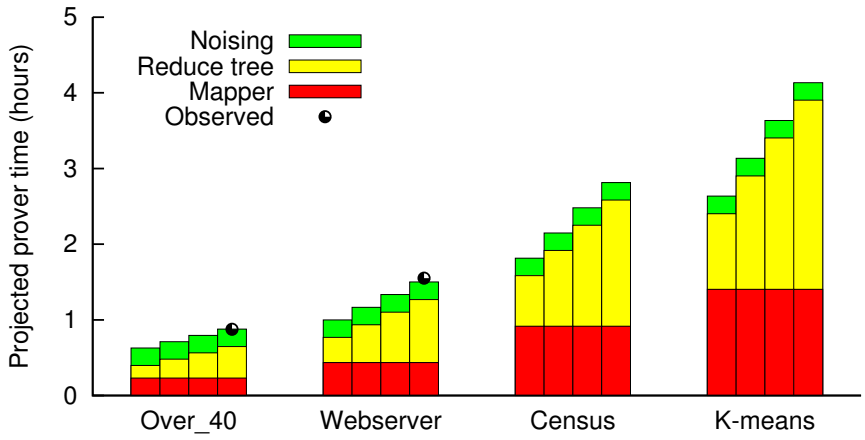


Figure 5.9: Projected time to generate proofs for databases of (8k, 16k, 32k, 62k) rows.

on Figure 5.9 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.

5.7.6 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 5.11 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

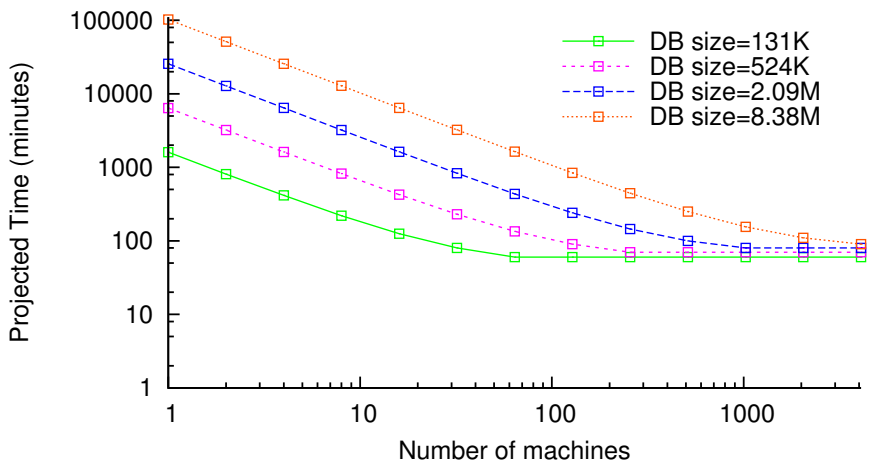


Figure 5.10: Estimated time to run the "over 40" query using a variable number of machines. Note log-log scale.

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.

5.7.7 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.

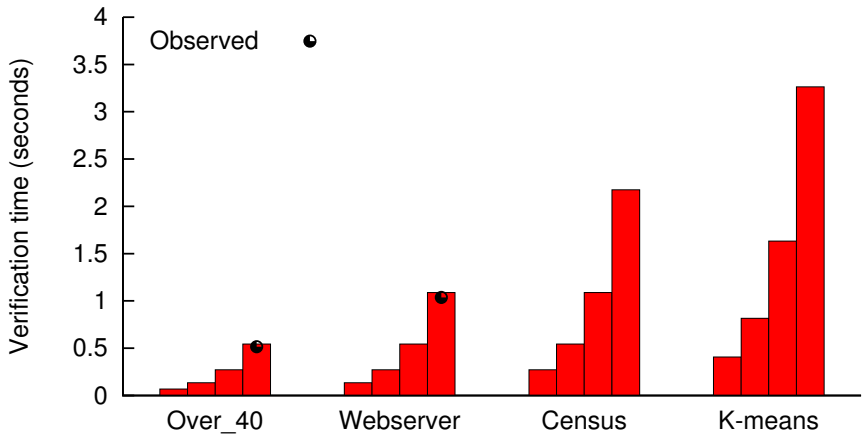


Figure 5.11: Projected time to verify complete programs for databases of (16k, 32k, 62k) rows on one machine.

Appendix: Proof of Privacy

Outline: Our goal is to prove that if a VFuzz program has the appropriate type, then the result is differentially private. Our proof is based on the proof in Reed and Pierce (2010) that all Fuzz programs that have a certain type are differentially private.

First, we will list the changes from Fuzz to VFuzz, and show that the Fuzz proof transfers in a straightforward way. Second, we show that the way we break down VFuzz programs into multiple separate Pantry programs does not affect our differential privacy guarantee. Finally, we show that our Pantry execution does not leak any additional information.

We briefly review the main privacy theorem statement from Fuzz(Reed and Pierce, 2010, §4): Executing a Fuzz program of type $!_n db \multimap \bigcirc\mathbb{R}$, and adding Laplace noise proportional to $n * \varepsilon$ is $\varepsilon-$differentially private.

Sketch: The Fuzz type system gives differential privacy through three theorems: progress and preservation (the standard type safety theorems), and a novel *metric preservation* theorem. Progress states that a well typed program will execute to completion: i.e. it will not get stuck. Type preservation states that a program with a given type will retain that type when executed. The metric preservation theorem

states that every execution of a well-typed function has at most the sensitivity that the type system assigns. The type notation $!_n\tau$ refers to the initial metric on τ "scaled up" by a factor of n.

Proposition : Executing a VFuzz program of type $!_n db \multimap \bigcirc\mathbb{R}$, and adding Laplace noise proportional to $n * \varepsilon$ is also $\varepsilon-$differentially private.

Proof: VFuzz differs from Fuzz in two ways: first, through the removal of Fuzz *terms*, and second, through the addition of some terms. Importantly, there are no additions, removals, or modifications to the Fuzz *type* language. The Fuzz type safety and metric preservation theorems are by induction(Reed and Pierce, 2010), and so removal of *terms* in VFuzz requires no additional proof burden, as we simply eliminate those inductive cases. The second way in which VFuzz differs from Fuzz is through adding terms for imperative functions, and assigning them types. We now prove that those terms do have the types we assign them in VFuzz.

VFuzz permits imperative functions in two situations: an imperative function as an argument to "map", with type $\mu \to$ `clipped` $\mathbb{R}$, (the clipped refers to a return value in the range $[0, R]$), and an imperative function as an argument to "split" with type $\mu \to bool$.

Consider the "split" in Fuzz(Reed and Pierce, 2010) without imperative functions: it has type `split` : $\tau\ bag \to (\tau \to bool) \multimap \tau\ bag$[4]. Note that the second argument is a function f of type $(\tau \to bool)$. Each invocation of f on a single row can influence the output by at most the inclusion or exclusion of a single row in the output of the whole split. This is exactly the differential privacy guarantee we seek, so we need no additional examination of the internal structure of the function. Thus, it suffices to verify that the output of f is a boolean. Thus, the result of a "split" in VFuzz with imperative functions retains the same type as in Fuzz.

Similarly, consider the "map" in Fuzz without imperative functions: it has type `map` : $\tau\ bag \to (\tau \to \sigma) \multimap \sigma\ bag$ from (Reed and Pierce, 2010). The second argument

[4]In Fuzz, "bag" is a multiset. Thus, the entire database is a multiset of rows, so if τ is the type of a row, the database has type τ bag

is a function f from τ to σ. In VFuzz, we restrict τ to type `row`, and σ to type `clipped` $\mathbb{R}$. Thus, we only need to check that the provided imperative function f has type `row` $\rightarrow$ `clipped` $\mathbb{R}$. As in Airavat (Roy et al., 2010), we can ensure this by truncating the return value of the function to the maximum allowed value before returning it.

Proposition: Splitting a single VFuzz program into separate `subprograms` each consisting of a single `sample` call retains the differential privacy guarantee.

Proof: A VFuzz program can consist of multiple `subprograms`. Each VFuzz subprogram is compiled into a single Pantry program. A subprogram is defined as a function (or nested series of functions) that ends in a single `sample` call. The type of a `sample` call in VFuzz is $\mathbb{R} \multimap \bigcirc\mathbb{R}$. Thus, a valid VFuzz subprogram must consist of a function, or a composed series of functions that has finite sensitivity with a return type of $\mathbb{R}$, followed by a `sample`. Since a VFuzz subprogram takes a database as input, each VFuzz subprogram has type $!_n db \multimap \bigcirc\mathbb{R}$.

A VFuzz program may aggregate the results of multiple subprograms and post-process the results, or simply print them individually. The post-processing that happens after sampling is public. By the composition theorem of differential privacy(McSherry and Talwar, 2007, §2.3), it suffices to examine each subprogram independently, and add up the epsilon costs to get a single ε_{total}-differentially private program.

Proposition: If a Pantry zero-knowledge proof leaks no information about the external block store, executing the VFuzz program in Pantry does not leak any additional information.

Proof: The VFuzz compiler does not have access to private data when it constructs the Pantry programs. Thus, the *structure* of the Pantry circuits do not contain any private information. We make no changes to the Pantry runtime, which guarantees that the Pinocchio proof leaks no information about the external block store. Since the private data set is only stored in the external block store of Pantry, none of the private data is leaked through the Pantry zero knowledge proofs.

5.8 Related work on verifiable computing

Verifiable computation: Although there has been significant theoretical work on proof-based verifiable computation for quite some time (see (Parno et al., 2013; Setty, Vu, Panpalia, Braun, Blumberg, and Walfish, 2012b; Vu, Setty, Blumberg, and Walfish, 2013) for surveys), efforts to create working implementations have only begun recently. These efforts have taken a number of different approaches. One set of projects (Cormode, Mitzenmacher, and Thaler, 2012; Thaler, Roberts, Mitzenmacher, and Pfister, 2012; Thaler, 2013), derived from interactive proofs as in Goldwasser, Kalai, and Rothblum (2008), 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 (Setty, McPherson, Blumberg, and Walfish, 2012a; Setty et al., 2012b; Setty, Braun, Vu, Blumberg, Parno, and Walfish, 2013; Vu et al., 2013) combines the interactive arguments of Ishai, Kushilevitz, and Ostrovsky (2007) with a compiler that supports program constructs such as branches, loops, and inequalities and an implementation that leverages GPU cryptography. Zaatar (Setty et al., 2013), the latest work in that series, exploits the constraint encoding of Gennaro, Gentry, Parno, and Raykova (2013) for smaller, more efficient proofs. This encoding is also used by Pinocchio (Parno et al., 2013), which offers similar functionality to Zaatar while supporting proofs that are both non-interactive and zero-knowledge. Pantry (Braun et al., 2013), 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 (Ben-Sasson, Chiesa, Genkin, Tromer, and Virza, 2013; Ben-Sasson, Chiesa, Tromer, and Virza, 2014; Wahby, Setty, Ren, Blumberg, and Walfish, 2014); 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 (Meiklejohn, Erway, Küpçü, Hinkle, and Lysyanskaya, 2010), ZQL (Fournet et al., 2013), and ZØ (Fredrikson and Livshits, 2014), 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 crowd-sourced 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.

5.9 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 (Parno et al., 2013) and Pantry (Braun et al., 2013) 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.

6

Conclusion

6.1 Summary

In this dissertation, we have looked at methods for increasing useful access to private datasets, even in the face of obstacles such as dispersed datasets, potentially malicious queriers, and the verifiability needs of modern scientific practices. However, today this data is justifiably locked away, and will not be available until individual privacy is provably safeguarded. In order to do so, we have studied three concrete real-world challenges that arise when providing provable differential privacy guarantees. We have designed and built systems to address each of these scenarios, and empirically evaluated their efficacy at solving these challenges.

In Chapter 3, we presented Fuzz, a runtime for *secure execution* of untrusted queries. In addition to the differential privacy guarantees on query outputs, Fuzz also protects against covert-channel attacks. Fuzz provides these guarantees through the use of a novel primitive called *predictable transactions*, which executes queries written in the differentially private programming language by Reed and Pierce (2010).

Finally, we showed some proof-of-concept attacks on existing differential privacy query systems such as PINQ and Airavat, and we experimentally verified that Fuzz's design effectively closed those channels, at the expense of a higher query completion time.

In Chapter 4, we described DJoin, a system that provides strong differential privacy guarantees while answering *distributed queries*. Unlike previous systems that focused only on horizontally partitioned databases, DJoin also supports *join* queries. DJoin uses two novel primitives to process join queries: BN-PSI-CA, a differentially private form of private set intersection cardinality, and DCR, a multi-party combination operator that can aggregate noised cardinalities without compounding the individual noise terms. Our experimental evaluation showed that DJoin can process realistic queries at practical timescales.

In Chapter 5, we presented VerDP, a system for private data analysis that provides both *verifiability*, as well as strong differential privacy guarantees. Differential privacy, while providing extremely strong guarantees about privacy, does so through adding noise to the answer: noise that a malicious querier can use to create plausible deniability. Current scientific practices rely on the ability to verify results, so differentially private studies would be hard to verify. VerDP solves this by providing verifiability through zero-knowledge proofs of execution, in addition to differential privacy guarantees. VerDP accepts queries written in a slightly modified version of the Fuzz query language which we call VFuzz. The VerDP runtime then produces both a differentially private output as in Fuzz, as well as a zero-knowledge proof of correctness, which allows independent third parties to verify that the answer was computed correctly, and that the noise was honestly chosen from the appropriate distribution in order to achieve privacy. Our experimental evaluation shows that VerDP successfully processed several different VFuzz queries that were functionally equivalent to Fuzz queries with minimal source code modification, and generated zero knowledge proofs in a distributed fashion with a nontrivial but nevertheless

affordable overhead.

6.2 Future Work

In this dissertation we have explored three particular directions for realizing distributed differential privacy implementations, but there are other directions that make sense for specific applications.

One example is in the design of Fuzz: while Fuzz uses a special type-system to only execute queries that are proven differentially private, the run-time is not *proven* side-channel free. In particular, it would be useful to prove the post-processing steps, and budget management private as well, such as in Ironclad (Hawblitzel et al., 2014).

A second example is in the design of DJoin. DJoin was the first distributed differential privacy runtime to process Join queries. However, we designed DJoin's optimizations to consider all participant databases' resources equally. One scenario where this might not be true would be when one database is vastly more private than another. For instance, if a user were joining a health-care database with another less private database, minimizing the privacy cost with respect to the health-care data would be more important.

We have already seen in DJoin that optimizing the structure of a distributed query can affect its runtime. For a more complex query language, optimizations also affect the total privacy budget used for a given query, and on which participant database this cost falls on. Users could use an economic model such as that described by Hsu et al. (2014) to measure the cost to a database of participating in a DJoin query, and then minimize the total privacy cost.

A second direction would be to make the existing systems faster. Both DJoin and VerDP have significant overheads in their execution: running times are measured in hours for real world queries, while the non-private versions take seconds. While this is significant, it is important to note that both DJoin and VerDP were the first to allow their respective queries to be executed at all, thus representing a significant

improvement over not having private access at all. However, it should still be possible to reduce those runtimes further. In the case of DJoin, one avenue for exploration would be to consider the existing database optimization literature. If a particular optimization can be proven differentially private, then DJoin could take advantage of it. VerDP can also take advantage of future advances in verifiable computing.

Ultimately, DJoin is a still limited to a small class of queries, and broadening distributed differential privacy to a broader set of queries remains an open problem. There are important real-world queries, such as Narayan, Papadimitriou, and Haeberlen (2014), that are beyond DJoin, but are still important to be able to answer in a differentially private fashion.

Bibliography

Alessandro Acquisti and Jens Grossklags. What can behavioral economics teach us about privacy. In *Digital Privacy: Theory, Technologies and Practices*, page 329. CRC Press, 2007. [Cited on page 6.]

Johan Agat. Transforming out timing leaks. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2000. [Cited on pages 17 and 51.]

Charu C Aggarwal. On k-anonymity and the curse of dimensionality. In *International Conference on Very Large Databases (VLDB)*, 2005. [Cited on page 7.]

Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE Symposium on Security and Privacy*, 2015. [Cited on page 50.]

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *ACM Conference on Computer and Communications Security (CCS)*, 2010. [Cited on pages 17 and 50.]

Mikhail J Atallah and Keith B Frikken. Securely outsourcing linear algebra computations. In *ACM Conference on Computer and Communications Security (CCS)*, 2010. [Cited on page 6.]

Michael Barbaro and Tom Zeller. A face is exposed for AOL searcher No. 4417749. *The New York Times*, August 2006. http://select.nytimes.com/gst/abstract.html?res=F10612FC345B0C7A8CDDA10894DE404482. [Cited on pages 91 and 92.]

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013. [Cited on page 108.]

Robert M Bell and Yehuda Koren. Lessons from the netflix prize challenge. *ACM SIGKDD Explorations Newsletter*, 2007. [Cited on pages 2 and 92.]

Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS)*, 2008. [Cited on pages 59, 64, 77, and 81.]

Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *International Cryptology Conference (CRYPTO)*, 2013. [Cited on page 121.]

Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, 2014. [Cited on page 121.]

John Bethencourt, Dawn Song, and Brent Waters. New constructions and practical applications for private stream searching (extended abstract). In *IEEE Symposium on Security and Privacy*, 2006. [Cited on page 78.]

Heidi Blake, Holly Watt, and Robert Winnett. 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`. [Cited on pages 87 and 91.]

Avrim Blum, Cynthia Dwork, Frank McSherry, and Kobbi Nissim. Practical privacy: the SuLQ framework. In *Principles of Database Systems (PODS)*, 2005. [Cited on pages 42 and 110.]

Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. Verifying computations with state. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013. [Cited on pages 88, 94, 96, 103, 121, and 122.]

Joseph A Calandrino, Ann Kilzer, Arvind Narayanan, Edward W Felten, and Vitaly Shmatikov. " you might also like:" privacy risks of collaborative filtering. In *IEEE Symposium on Security and Privacy*, 2011. [Cited on page 8.]

CDC and HHS. HIPAA privacy rule and public health. Guidance from CDC and the US Department of Health and Human Services. *MMWR: Morbidity and Mortality Weekly Report*, 2003. [Cited on page 1.]

Shuchi Chawla, Cynthia Dwork, Frank McSherry, Adam Smith, and Hoeteck Wee. Toward privacy in public databases. In *Theory of Cryptography Conference (TCC)*, 2005. [Cited on pages 42 and 110.]

Ruichuan Chen, Alexey Reznichenko, Paul Francis, and Johannes Gehrke. Towards statistical queries over distributed private user data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012. [Cited on pages 13, 54, and 58.]

Ruichuan Chen, Istemi Ekin Akkus, and Paul Francis. SplitX: high-performance private analytics. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2013. [Cited on page 14.]

Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *ACM Innovations in Theoretical Computer Science (ITCS)*, 2012. [Cited on page 121.]

Scott Crosby, Dan Wallach, and Rudolf Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security*, 2009. [Cited on page 17.]

Brian Deer. MMR doctor Andrew Wakefield fixed data on autism. *The Sunday Times*, February 8, 2009. `http://www.thesundaytimes.co.uk/sto/public/news/article148992.ece`. [Cited on pages 87 and 91.]

Yitao Duan, John Canny, and Justin Zhan. P4P: Practical large-scale privacy-preserving distributed computation robust against malicious users. In *USENIX Security Symposium*, 2010. [Cited on page 14.]

Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2006. [Cited on pages 50 and 72.]

Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation (TAMC)*, 2008. [Cited on pages 2 and 50.]

Cynthia Dwork. The differential privacy frontier (extended abstract). In *Theory of Cryptography Conference (TCC)*, 2009. [Cited on page 50.]

Cynthia Dwork and Moni Naor. On the difficulties of disclosure prevention in statistical databases or the case for differential privacy. *Journal of Privacy and Confidentiality*, 2008. [Cited on page 8.]

Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *European Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2006a. [Cited on pages 15, 29, 54, 58, 59, 62, 65, 77, and 107.]

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*, 2006b. [Cited on pages 2, 10, 42, 100, and 110.]

Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010. [Cited on page 56.]

Cedric Fournet, Markulf Kohlweiss, George Danezis, and Zhengqin Luo. ZQL: A compiler for privacy-preserving data processing. In *USENIX Security Symposium*, 2013. [Cited on pages 92 and 122.]

Matthew Fredrikson and Benjamin Livshits. ZØ: An optimizing distributing zero-knowledge compiler. In *USENIX Security Symposium*, 2014. [Cited on page 122.]

Michael Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *European Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2004. [Cited on pages 55, 56, 59, 60, 61, and 77.]

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013. [Cited on pages 87, 96, and 108.]

Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems*, May 2001. [Cited on pages 18 and 51.]

Simson Garfinkel. *PGP: pretty good privacy*. O'Reilly Media, Inc., 1995. [Cited on page 6.]

James Garrity and Murat Kantarcioglu. UTD Paillier threshold encryption toolbox. Available from `http://utdallas.edu/~mxk093120/paillier/`, 2012. [Cited on page 77.]

Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *European Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013. [Cited on page 121.]

Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. In *ACM Symposium on the Theory of Computing (STOC)*, 2008. [Cited on page 121.]

Michaela Götz and Suman Nath. Privacy-aware personalization for mobile advertising. Technical Report MSR-TR-2011-92, Microsoft Research, 2011. [Cited on pages 54 and 58.]

Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. Differential privacy under fire. In *USENIX Security Symposium*, 2011. [Cited on pages iii and 96.]

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014. [Cited on pages 52, 107, and 125.]

Thomas Herndon, Michael Ash, and Robert Pollin. 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, 2013. [Cited on page 86.]

Seth Hettich and Steven D. Bay. The UCI KDD archive. Univ. of California Irvine, Dept. of Information and Computer Science, http://kdd.ics.uci.edu/, 2015. [Cited on page 44.]

Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C. Pierce, and Aaron Roth. Differential privacy: An economic method for choosing epsilon. In *IEEE Conputer Security Foundations Symposium (CSF)*, 2014. [Cited on pages 10 and 125.]

Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy*, 1991. [Cited on pages 17 and 50.]

ICPSR. ICPSR Data Deposit at the University of Michigan, 2015. http://www.icpsr.umich.edu/icpsrweb/deposit/. [Cited on page 89.]

iDASH. Integrating Data for Analysis, Anonymization and SHaring, 2015. http://idash.ucsd.edu/. [Cited on page 89.]

Jeneen Interlandi. An unwelcome discovery. *The New York Times*, October 22, 2006. www.nytimes.com/2006/10/22/magazine/22sciencefraud.html. [Cited on pages 87 and 91.]

IPUMS. Integrated Public Use Microdata Series at the Minnesota Population Center, 2015. https://www.ipums.org/. [Cited on page 89.]

Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *IEEE Conference on Computational Complexity (CCC)*, 2007. [Cited on page 121.]

Myong H. Kang, Ira S. Moskowitz, and Daniel C. Lee. A network pump. *Transactions on Software Engineering*, 1996. [Cited on pages 17 and 50.]

Lea Kissner and Dawn Song. Privacy-preserving set operations. In *International Cryptology Conference (CRYPTO)*, 2005. [Cited on pages 55, 56, 59, 60, 63, 70, and 77.]

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *International Cryptology Conference (CRYPTO)*, 1999. [Cited on pages 18 and 51.]

Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *International Conference on Very Large Databases (VLDB)*, 1986. [Cited on page 74.]

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 1973. [Cited on pages 17 and 50.]

Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA, 1990. [Cited on pages 18 and 38.]

Xavier Leroy and Damien Doligez. Caml Light website. http://caml.inria.fr/caml-light/index.en.html, 2004. [Cited on pages 18 and 38.]

Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004. [Cited on page 56.]

Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7:76, 2003. [Cited on page 2.]

Lothar F Mackert and Guy M Lohman. R* optimizer validation and performance evaluation for distributed queries. In *International Conference on Very Large Databases (VLDB)*, 1986. [Cited on page 57.]

Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michae Walfish. Depot: Cloud storage with minimal trust. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010. [Cited on page 56.]

Jonathan R Mayer and John C Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy*, 2012. [Cited on page 2.]

Andrew McGregor, Ilya Mironov, Toniann Pitassi, Omer Reingold, Kunal Talwar, and Salil Vadhan. The limits of two-party differential privacy. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2010. [Cited on page 12.]

Frank McSherry. Privacy integrated queries. In *ACM International Conference on Management of Data (SIGMOD)*, 2009. [Cited on pages 13, 16, 20, 26, 50, 54, 58, 70, 72, 93, and 108.]

Frank McSherry and Ilya Mironov. Differentially private recommender systems: Building privacy into the net. In *ACM Conference on Knowledge Discovery and Data mining (KDD)*, 2009. [Cited on page 16.]

Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2007. [Cited on pages 70 and 120.]

Sarah Meiklejohn, C. Chris Erway, Alptekin Küpçü, Theodora Hinkle, and Anna Lysyanskaya. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security Symposium*, 2010. [Cited on page 122.]

Ilya Mironov. On significance of the least significant bits for differential privacy. In *USENIX Security Symposium*, 2012. [Cited on pages 49, 52, and 107.]

Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In *International Cryptology Conference (CRYPTO)*, 2009. [Cited on page 12.]

Arjun Narayan and Andreas Haeberlen. Djoin: Differentially private join queries over distributed databases. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012. [Cited on page iii.]

Arjun Narayan, Antonis Papadimitriou, and Andreas Haeberlen. Compute globally, act locally: Protecting federated systems from systemic threats. In *Workshop on Hot Topics in System Dependability (HotDep)*, 2014. [Cited on page 126.]

Arjun Narayan, Antonis Papadimitriou, and Andreas Haeberlen. Verifiable differential privacy. In *ACM European Conference on Computer Systems (EuroSys)*, 2015. [Cited on page iii.]

Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, 2008. [Cited on pages 2, 7, 91, and 92.]

C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *ACM Conference on Computer and Communications Security (CCS)*, 2001. [Cited on page 63.]

Margaret L O'Donnell. FERPA: Only a piece of the privacy puzzle. *Journal of College and University Law*, 29:679, 2002. [Cited on page 1.]

Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *European Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999. [Cited on page 60.]

Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013. [Cited on pages 93, 94, 121, and 122.]

Benny Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explorations Newsletter*, 4(2), 2002. [Cited on page 56.]

pinq. PINQ website, 2015. `http://research.microsoft.com/en-us/projects/pinq/`. [Cited on page 26.]

Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011. [Cited on page 56.]

Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and countermeasures for smart cards. In *Proceedings of the International Conference on Research in Smart Cards (E-SMART)*, September 2001. [Cited on pages 18 and 51.]

Vibhor Rastogi and Suman Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *ACM International Conference on Management of Data (SIGMOD)*, 2010. [Cited on pages 54 and 58.]

Jason Reed and Benjamin C Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *ACM International Conference on Functional Programming (ICFP)*, 2010. [Cited on pages 18, 31, 33, 35, 87, 95, 96, 97, 118, 119, and 123.]

Indrajit Roy, Srinath Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010. [Cited on pages 13, 16, 20, 27, 50, 54, 58, 70, 72, 93, and 120.]

Srinath Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *Network and Distributed System Security Symposium (NDSS)*, 2012a. [Cited on page 121.]

Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, 2012b. [Cited on page 121.]

Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *ACM European Conference on Computer Systems (EuroSys)*, 2013. [Cited on page 121.]

Elaine Shi, T-H Hubert Chan, Eleanor G Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *Network and Distributed System Security Symposium (NDSS)*, 2011a. [Cited on page 15.]

Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *Network and Distributed System Security Symposium (NDSS)*, 2011b. [Cited on page 58.]

Paritosh Shroff and Scott F. Smith. Securing timing channels at runtime. Technical report, The Johns Hopkins University, July 2008. [Cited on pages 17 and 51.]

Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 2002. [Cited on page 7.]

Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *International Cryptology Conference (CRYPTO)*, 2013. [Cited on page 121.]

Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2012. [Cited on page 121.]

THEP. The Homomorphic Encryption Project, 2012. http://code.google.com/p/thep/. [Cited on page 77.]

Jaideep Vaidya and Chris Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 2005. [Cited on pages 55, 56, 59, and 60.]

Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013. [Cited on page 121.]

Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. Cryptology ePrint 2014/674, 2014. [Cited on page 121.]

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem. *ACM Transactions on Embedded Computing Systems*, 2008. [Cited on page 51.]

Michael Wilson, Ron Cytron, and Jonathan Turner. Partial program admission. In *IEEE Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, 2009. [Cited on page 51.]

J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, 1991. [Cited on pages 17 and 50.]

Andrew Yao. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982. [Cited on pages 15, 54, 59, and 64.]

Ning Zhang, Ming Li, and Wenjing Lou. Distributed data mining with differential privacy. In *IEEE International Conference on Communications*, 2011. [Cited on page 56.]