Position Paper: Differential Privacy with Information Flow Control

Arnar Birgisson
Chalmers University of Technology
arnar.birgisson@chalmers.se

Frank McSherry Martín Abadi
Microsoft Research
{mcsherry,abadi}@microsoft.com

Abstract

We investigate the integration of two approaches to information security: information flow analysis, in which the dependence between secret inputs and public outputs is tracked through a program, and differential privacy, in which a weak dependence between input and output is permitted but provided only through a relatively small set of known differentially private primitives.

We find that information flow for differentially private observations is no harder than dependency tracking. Differential privacy's strong guarantees allow for efficient and accurate dynamic tracking of information flow, allowing the use of existing technology to extend and improve the state of the art for the analysis of differentially private computations.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

General Terms Languages, Security

Keywords Differential Privacy, Information Flow Control

1. Introduction

Understanding the flow of sensitive information through computer programs, and the implications the visible outcomes may have about the private inputs, remains a challenge for information security. One particular challenge on which we focus relates to aggregate information, data about a large number of secret inputs (perhaps individual medical records) but reduced in some way which intuitively provides some qualitative protection. Many standard approaches to information flow control [13] rightly conclude that the results of such computations depend on all sensitive inputs, and that their disclosure could reveal information about each of those inputs.

One promising approach to enabling privacy-preserving use of individual data is *differential privacy* [6, 7]. Differential privacy provides a data analyst (or their analysis) with access to certain aggregates of sets of possibly sensitive records. To limit the analyst's ability to draw inferences about individual records, the aggregate results are carefully randomized. By adding an appropriately distributed random value to the exact result, one can bound the increase or decrease in the relative probability of a particular randomized result, when one individual's record is added to or removed from the dataset. This bound restricts the amount of information that the release of the output may reveal about any one input.

Our interest in this note is to study the connection between differential privacy and information flow. In particular, as differential privacy acts as a quantitative measure for the information disclosed through one quantity (the output) about another (any of the secret inputs), it seems possible to use it as a form of quantitative information flow control. As we will see, the strong compositional properties of differential privacy allow us to reason easily about the differential privacy cost of a sub-computation based only on the differentially private measurements it uses, and not at all on how they are used.

The main limitation of differential privacy appears to be that most traditional programs simply do not satisfy it. Prototypical examples such as the password program, emitting true if and only if the input equals a secret value, simply do not have any quantitative bound in the differential privacy framework. Programs compatible with differential privacy are typically built out of differentially private primitive computations — many standard aggregations exist, and more are being developed as the research progresses. Our goal then, is not to replace quantitative information flow with differential privacy, nor vice versa, but to explore the combination of differential privacy with information flow control. We find that this combination is both viable and beneficial.

1.1 Programming with Differential Privacy

As mentioned in the previous section, differentially private programs are so almost always because they access the underlying secret inputs through well understood and standard primitives. These primitives are typically aggregations run across a collection of inputs, though advanced programming languages [10–12] are enriching this space. Each computation provides a quantitative bound on the probabilistic dependence between the output value and the secret inputs, often referred to as the "differential privacy cost".

Most differentially private computations will make many measurements of the underlying data, each costing privacy and increasing the quantitative flow of information through the program. One virtue of differential privacy is that the composition rules are so easy: the costs of multiple measurements simply add up. However, as the underlying measurement substrates have no understanding of how the differentially private measurements are being used, they are typically configured to interrupt access to a dataset when a certain quantitative bound is breached.

While this behavior is safe, it can be overly conservative. In many such computations, the analyst is not the final consumer of the computation; policy makers, implementors, and other analysts may consume strict subsets of the results, and even integrate them with the results of other differentially private computations. For a complex computation built out of uncoordinated parts, understanding the actual differential privacy cost can allow both more liberal disclosure as well as provide more realistic privacy guarantees.

Concretely, imagine a decision tree classifier for medical diagnosis. Each node in the decision tree is based on distinct measurements of some sensitive training data (actual medical records), but it is only when the new patients arrive and the tree is used that we discover which nodes are required and how much information about the source data is disclosed. Different hospitals (and units within a hospital) may end up using very different sub-trees to classify their patients, and understanding how much information is disclosed to each party (patient, unit, hospital, and the world) is both challenging and important.

1.2 Contributions

We present a theoretical framework, based on the call-by-name λ-calculus, which allows programs using differentially private measurements to reveal their actual dependence on their underlying inputs through the measurements essential to their evaluation. We rely on lazy execution, so that queries for information that is not needed in the final result are never performed, even if they are part of the code. We rely on lazy execution, so that queries for information that is not needed in the final result are never performed, even if they are part of the code; further, we collect dependency information dynamically (much as in [3]). We prove that our framework maintains the statistical guarantee of differential privacy, i.e., that programs written in this framework are themselves differentially private given that the underlying mechanism is.

The rest of this paper is organized as follows. Section 2 presents the necessary background on differential privacy. Section 3 defines our formal language and proves that the differential privacy guarantee is maintained. Section 4 discusses a prototype implementation in Haskell. Section 5 concludes.

1.3 Related Work

This work provides the combination of three main features: differential privacy, dynamic tracking, and quantitative information flow. In the absence of any one of the three, we find influential related work.

Putting aside differential privacy, we have the problem of dynamic quantitative information flow, on which a substantial amount of work exists [8, 9]. One of the main challenges in this work is the ability to deal with aggregate or otherwise anonymized data. There is little basis for declassification without a privacy guarantee like differential privacy, and guarantees other than differential privacy appear to lack the strong properties which align it with program analysis. The same challenges apply to static methods for quantitative information flow, which has been well studied. See e.g. [13] for an overview.

If we do not require dynamic analysis, previous work of Reed and Pierce [11] provides a static type system in which one can express metric properties of programs, including differential privacy parameters. While this approach is promising, its static nature forces it to be pessimistically conservative on issues of control flow. Conditional statements are given the maximum possible cost of their branches; loops and recursion can be typed only if their body has no privacy cost; loops or recursion with non-trivial measurements are typed as having infinite privacy cost. By dynamically checking programs we achieve greater accuracy, and by virtue of differential privacy's guarantees they do not leak information through the analysis's execution.

If we do not employ information flow techniques, at least two previous works apply dynamic accounting to differentially private computation: Privacy Integrated Queries (PINQ) [10] and Airavat [12] each provide a query language in which each query has statically determinable privacy guarantees, and programs whose data access is limited to these mechanism inherit their differential privacy guarantees. However, the two approaches do nothing more than accounting; if a computation does not make full use of the measurements made, the systems are unable to acknowledge the reduced privacy cost.

Other recent related work compares differential privacy with information-theoretic notions and with quantitative information flow [4, 5]. Alvim et al. consider the connections between differential privacy and definitions of quantitative information leaks, focusing on quantifications of mutual information between an attacker's knowledge before and after observing a program. They conclude that differential privacy provides stronger bounds on leakage. In [5], Barthe and Köpf show an upper bound for information leaked by differentially private programs. They show that the bound must depend on the size of the input domain for secrets. Our work differs in that we do not establish these theoretical connections, but instead focus on improving the programming model for differentially private programs.

2. Background

The technical definition of differential privacy states that the distribution of the outputs of a randomized computation changes only slightly if its inputs change a little. We consider computations that depend on *data sets*, collections of records. We use the operator $\oplus$ to represent the symmetric difference of two data sets: $A \oplus B$ is the collection of records appearing in exactly one of of the data sets A and B.

Definition 1. *We let C be a random variables representing the outcome of a randomized computations depending on data sets. We say that C is* ϵ-differentially private*, if for any data sets A and B, and any set $S \subseteq Range(C)$, the following inequality holds:*

$$\Pr[C \in S \mid A] \leq \Pr[C \in S \mid B] \cdot \exp(\epsilon \cdot |A \oplus B|)$$

While it is useful in the above definition to think of computations and their outcomes, we can replace $C \in S$ with any probabilistic event which may depend on a dataset.

When multiple computations are composed into a larger one, each computation depending on the same input dataset, we would like to say something about the privacy of the composed computation. By managing the noise added to each individual query made on the dataset, it is possible to provide differential privacy guarantees about the composed result. PINQ [10] provides a mechanism that ensures that rich collections of queries together adhere to differential privacy. We call such a mechanism a *differentially private mechanism*, i.e., one that gives the above guarantee for any collection of queries, as well as each individual one.

Definition 2. *Let M be a random variable representing a partial function, mapping* measurements m *(which shall remain abstract) to values, depending on an underlying data set. We say that M is a* differentially private mechanism *if for any collection of measurements $m_1, \dots, m_n$, each with a specified privacy cost $\epsilon_m \in \mathbb{R}$, the following holds for any two datasets A and B:*

$$\begin{aligned} &\Pr[\forall i : M(m_i) \in S_i \mid A] \\ \leq &\Pr[\forall i : M(m_i) \in S_i \mid B] \cdot \exp(\epsilon \cdot |A \oplus B|) \end{aligned}$$

where $\epsilon = \sum_i \epsilon_i$.

In particular, if a computation depends on a set of measurements obtained from a differentially private mechanism, but is deterministic otherwise, then it is differentially private according to Definition 1.

$$\begin{array}{rcll} x, y & \in & \mathsf{Vars} & \text{Variables} \\ m, n & \in & \mathsf{MVars} & \text{Measurement vars} \\ t & ::= & [m \mapsto v] & \text{Tags} \\ e & ::= & x \mid m \mid \lambda x.e & \text{Terms} \\ & & \mid e\ e \mid t : e & \\ u, v & ::= & \lambda x.e \mid t : v & \text{Values} \\ E & ::= & [\cdot] \mid E\ e \mid t : E & \text{Evaluation contexts} \end{array}$$

Figure 1. Syntax and evaluation contexts

$$\textsc{ctx}\ \frac{e \to_M e'}{E[e] \to_M E[e']} \qquad \beta\ \frac{}{(\lambda x.e_1)\ e_2 \to_M e_1[e_2/x]}$$

$$M\ \frac{M(m) = v}{m \to_M [m \mapsto v] : v} \qquad \textsc{t-lift}\ \frac{}{(t : e_1)\ e_2 \to_M t : (e_1\ e_2)}$$

Figure 2. Semantics

The main property of a differentially private mechanism is its ability to produce observations that are certified not to reveal too much about the source data sets.

3. Theory

The syntax of our language is presented in Figure 1. It is the untyped λ-calculus extended with terms that represent values to be acquired from an external mechanism, called measurement variables, and tagged terms. Tags are intended to indicate that an expression may depend on a value previously obtained from an external mechanism. Tagged values are also considered to be values. A tag describes both the measurement variable on which the subexpression may depend as well the value with which the measurement variable was instantiated.

For example, assume that we have the usual encodings of natural numbers and $+$. Then the expression $[m \mapsto 2] : 5$ indicates that the value 5 was computed using a measurement variable m, which was instantiated to 2 by some external mechanism. One possible program leading to this value is $m + 3$.

Tags represent *may*-dependencies. For example the tagged value might also be produced by the program `if true then 5 else m` (assuming also standard encodings of conditionals and booleans.) One of our goals will be to minimize the reporting of *false dependencies* by making use of a call-by-name evaluation order.

For two partial functions f, g we write $f \leq g$, and say f is a restriction of g, iff whenever $f(x)$ is defined it is equal to $g(x)$. However, g may be defined on points where f is not.

Figure 2 shows the small step semantics of our language. Evaluation contexts (Figure 1) together with the congruence rule CTX allow evaluation in the left-hand side of applications as well as inside tagged expressions. The evaluation relation is parametrized by a *measurement map* M.

Definition 3. *A* measurement map *is a partial function that maps measurement variables to closed, tag-free values.*

Besides the standard rule for call-by-name β-reduction we introduce two rules for managing tags. The M rule instantiates a measurement variable according to the measurement mechanism and tags the resulting value appropriately. The T-LIFT rule lifts tags appearing in the left-hand side of application, in order to obtain a term suitable for application of the β-rule.

We want a theorem that says that, after evaluating a term, we obtain a value whose tags describe the measurement maps that could have led to this evaluation. First we give some useful definitions and lemmas.

Definition 4. *Let M be a measurement map and m a measurement variable. The measurement map $M - m$ is defined by removing the definition of $M(m)$:*

$$(M - m)(n) = \begin{cases} M(x) & \textit{if } n \neq m \\ \text{undefined} & \textit{if } n = m \end{cases}$$

Definition 5. *A measurement variable m appears in a* persistent position *in an expression e iff*

- *$e = m$, or*
- *$e = [m \mapsto v] : e'$ for some v and e', or*
- *$e = t : e'$ or $e = e'\, e''$ and m appears in a persistent position in e'.*

Similarly, m appears in a persistent position in a context E iff E has the form $[m \mapsto v] : E$ for some v, or when it has the form $t : E'$ or $E'\, e$ for some e and m is in a persistent position in E'.

Lemma 1. *1. If m is in a persistent position in e, it is in a persistent position in $E[e]$.*
2. If m is in a persistent position in $E[e]$, it is in a persistent position in either E or e.

Proof. Induction on the structure of evaluation contexts. □

Note that if m is in a persistent position in E then it is in a persistent position in $E[e'']$ for any e''.

Lemma 2. *1. If $e \rightarrow_M e'$ and m is in a persistent position in e, then m is in a persistent position in e'.*
2. If $e \rightarrow^*_M e'$ and m is in a persistent position in e, then m is in a persistent position in e'.

Proof. The proof of part 1 is by induction on the derivation of $e \rightarrow_M e'$ with a case for each rule of the semantics.

- β-rule. e has the form $(\lambda x.e_1)\, e_2$. If m were to appear in a persistent position in e, it would have to do so in $\lambda x.e_1$, but abstractions cannot have any measurement variables in a persistent position. Therefore, this case is vacuous.
- M-rule. e has the form n. Since m is in a persistent position in e, we must have $n = m$. Therefore, $e' = [m \mapsto v] : v$ where $M(m) = v$, so m is in a persistent position in e'.
- T-LIFT rule. e has the form $(t : e_1)\, e_2$. Suppose m is in a persistent position in e. This can be the case only if $t = [m \mapsto v]$ for some v, or if m is in a persistent position in e_1. In the first case m is in a persistent position in $e' = t : (e_1\, e_2)$. In the second case m is in a persistent position in $(e_1\, e_2)$ and therefore also in $t : (e_1\, e_2)$.
- CTX-rule. This case follows directly from the induction hypothesis and part 2 of Lemma 1.

Part 2 follows directly from part 1 by induction. □

Lemma 3. *Let m be a measurement variable. If m does not appear in a persistent position in an expression e and $e \rightarrow_M e'$ for some M, then $e \rightarrow_{M-m} e'$.*

Proof. By induction on the derivation of $e \rightarrow_M e'$.

- β-rule. This rule does not depend on the measurement map, so if $e \rightarrow_M e'$ by it, then $e \rightarrow_N e'$ also for any N, in particular for $M - m$.
- M-rule. For this rule to apply, e must be m itself or some other measurement variable m'. The former case is impossible, for otherwise m would be in a persistent position in e. In the latter case, e' is $[m' \mapsto v] : v$ for some v such that $M(m') = v$, so $e \rightarrow_{M-m} e'$.
- T-LIFT rule. This rule does not depend on the measurement map. Since $e \rightarrow_M e'$ according to it, then $e \rightarrow_N e'$ for any measurement map N, in particular $M - m$.
- CTX-rule. This case follows directly from the induction hypothesis and part 1 of Lemma 1. □

The previous lemma easily extends to program runs.

Lemma 4. *If $e \rightarrow^*_M e'$ and m does not occur in a persistent position in e', then $e \rightarrow^*_{M-m} e'$.*

Proof. The claim is proved by induction on the length of the derivation $e \rightarrow^*_M e'$.

If this length is 0, the claim is trivial. Assume that $e \rightarrow^*_M e'' \rightarrow_M e'$ for some e''. By part 1 of Lemma 2, m does not occur in a persistent position in e''. Therefore by the induction hypothesis, $e \rightarrow^*_{M-m} e''$. Moreover, since m does not occur in a persistent position in e'', Lemma 3 gives that $e'' \rightarrow_{M-m} e'$, so by transitivity $e \rightarrow^*_{M-m} e'$. □

The above allows us to deduce that any tag that does not appear in a persistent position in the result of a derivation is not relevant for the derivation. Hence the evaluation does not need to instantiate the corresponding measurement (by invocation of the M-rule).

If we start from an expressions that contains no tags, the tags in the final result will tell us what parts of M the evaluation depends on. Any measurement map that agrees on the measurements described by the tags in the final value will give the same result.

Definition 6. *Let e be an expression. If any two tags with the same measurement variable also hold the same value we say that the tags in e are non-conflicting. If they are, then we let M_e be the measurement map generated by this set of tags.*

$$M_e(m) = \begin{cases} v & \text{if } [m \mapsto v] \text{ appears in } e \\ \text{undefined} & \text{otherwise} \end{cases}$$

Theorem 1. *Assume that e contains no tags. If $e \to_M^* v$ then*

$$e \to_{M'}^* v \quad \text{for all } M' \text{ such that } M_v \leq M'$$

Proof. First we need to show that M_v is well defined, i.e., that the tags in v are non-conflicting. Since e has no tags, all tags in v are produced by a $\to_M$ transition. The only rule that can produce tags is the M rule, so the value of any tag for a measurement variable m will be $M(m)$.

A tag does not appear in a persistent position in v if it does not appear in v at all. By iteratively applying Lemma 4 we obtain that $e \to_{M_v}^* v$. By induction on the derivation sequence we obtain that any measurement map M' such that $M_v \leq M'$ produces the same derivation. □

The above theorem states that the set of measurement maps that extend M_v is a subset of the set of measurement maps that cause e to evaluate to the value v. Furthermore, since v encodes the information about all measurements taken, the converse is true as well.

Theorem 2. *Given a tag-less expression e and a value v, the following set equivalence holds:*

$$\{M : e \to_M^* v\} = \{M : M_v \leq M\}$$

We omit the full proof for this theorem. Its key insight is that tags are introduced only by applications of the M-rule in the derivation, which obtains the tag from M, and that other rules do not modify the tag set of an expression.

We are now ready to make the connection with differential privacy. Theorem 1 describes a set of measurement maps that produce a given result. As evaluation is otherwise deterministic, the probability of an expression e evaluating to a certain value v is completely determined by the probability of the measurement map under which it is evaluated. An evaluation $e \to_M^* v$ may be viewed as a probabilistic event, determined by the probability with which M is realized by the privacy mechanism as the measurement map. We will use this observation to talk about differential privacy of evaluations, replacing $C \in S$ in Definition 1 with an evaluation.

Theorem 3. *The evaluation of an expression e given by $e \to_M^* v$ is ϵ-differentially private if M is a differentially private mechanism, where $\epsilon = \sum_i \epsilon_{m_i}$ for measurements m_i appearing in the tags of v.*

Proof. Assume M is a differentially private mechanism, and that $e \to_M^* v$. We want to show that

$$\Pr[e \to_M^* v \mid A] \leq \Pr[e \to_M^* v \mid B] \cdot \exp(\epsilon \cdot |A \oplus B|)$$

By Theorem 2 we have the following.

$$\Pr[e \to_M^* v \mid A] = \Pr[M_v \leq M \mid A]$$

Let $([m_i \mapsto v_i])_{i \in I}$ be the set of tags of v, determining M_v. The right-hand side of the above can be written

$$\Pr[\forall i \in I : M(m_i) = v_i \mid A]$$

As $\epsilon = \sum_i \epsilon_i$ and M is a differentially private mechanism, according to Definition 1 we have:

$$\begin{aligned} &\Pr[\forall i \in I : M(m_i) = v_i \mid A] \\ &\quad \leq \Pr[\forall i \in I : M(m_i) = v_i \mid B] \cdot \exp(\epsilon \cdot |A \oplus B|) \end{aligned}$$

which, again according to Theorem 2, equals

$$\Pr[e \to_M^* v \mid B] \cdot \exp(\epsilon \cdot |A \oplus B|).$$

Summarizing, we obtain

$$\Pr[e \to_M^* v \mid A] \leq \Pr[e \to_M^* v \mid B] \cdot \exp(\epsilon \cdot |A \oplus B|)$$

i.e., the evaluation is ϵ-differentially private. □

The importance of Theorem 3 is that it allows us to determine the differential privacy cost of an evaluation by looking only at the measurements present in tags of the resulting value. This result concretely connects the quantitative tracking of differentially private information with dependence tracking, an arguably much simpler task.

4. Implementation

We have experimented with prototype implementations of our ideas within a few different programing languages. Our main goal is to make writing general-purpose programs easier, using measurement mechanisms such as PINQ, while avoiding the need to annotate or rewrite useful existing code.

Our main approach is to use lazy evaluation to identify the essential measurements required from the underlying privacy mechanism to evaluate an expression. However, the languages we considered exhibited a tension between lazy semantics and transparency of evaluation: lazy languages were less inclined to violate referential transparency, concealing whether a measurement was truly needed for an evaluation, while strict languages such as F# [1] make it cumbersome to write lazy programs.

We were somewhat successful with an implementation in Haskell [2], a pure functional language that uses lazy evaluation by default. However, Haskell's referential transparency makes it non-trivial to *observe* the steps of evaluation, since these are (rightly) left up to the compiler rather than being fully specified by the language itself.

Monadic programming could be used to provide a very natural programming model for working on top of a differentially private database. However, this style of programming requires the programmer to be mindful of not "unpacking" values with the `<-` operator, unless they are strictly needed. In other words, we partly lose the benefits of lazy evaluation.

Therefore, we represent differentially private computations as pure computations of the following type:

```
type DPComp q r v = (q -> r) -> v
```

Such a computation depends on a function of type `q -> r`, representing the differentially private mechanism that turns queries of type `q` into results of type `r`. The computation uses the mechanism to produce a value of type `v`. With this type, invocations of the underlying mechanism are considered pure and will be performed only by need.

As said above, referential transparency does not allow us to observe these invocations, at least not while remaining fully pure. We surmount this obstacle by running a computation of type `DPComp q r v` with a `q -> r` function that throws an exception whenever it is invoked. The exception contains a continuation for proceeding with the computation once a result value is supplied. We then run the whole computation in a harness that catches the exception, calls the actual user-supplied `q -> r` function, logging both the query and the result, and then simply invokes the continuation to proceed. The implementation uses `unsafePerformIO` to escape the side-effects of exception handling and logging. Doing so is safe since the semantics of the pure part of the computation remains the same.

The harness hides all of these details, so the user needs only specify the computation itself, of type `DPComp q r v`, and a function acting as the differentially private mechanism of type `q -> r`. A variant of the harness can even accept a query function of the type `q -> IO r`, which may be appropriate in many cases. However, its invocations will be wrapped in `unsafePerformIO`, so the this function is responsible for maintaining apparent referential transparency.

Our experiments indicate that the framework described here works in principle, and is able to correctly report privacy costs while still conservatively issuing queries.

5. Conclusion

We have considered differential privacy as a candidate for quantitative information flow control. While differential privacy may be introduced only through limited interfaces to data, once introduced it becomes a robust quantification of information flow. In particular, one can determine the differential privacy parameter of a computation by determining only on which differentially private measurements the computed value depends, and accumulating their privacy costs.

Reasoning about the flow of differentially private information appears to be relatively simple. We have introduced a slight extension of the untyped lambda calculus to reflect dependence on exogenous measurements, and shown that the probability of derivation varies at most as much as the probability of the measurements. Consequently, differential privacy guarantees of the essential measurements propagate to computations derived from them.

We expect this line of research, and more careful analysis of the flow of differentially private information, will lead to more realistic systems for private data analysis. Our preliminary investigation of the implementation of such a system revealed that dependence tracking need not be complex. However, there is clearly more to do in this direction.

Acknowledgements Arnar Birgisson worked on this research while at Microsoft Research. Martín Abadi is also affiliated with UC Santa Cruz and Collège de France. We thank Andrei Sabelfeld and Úlfar Erlingsson for discussions and comments.

References

[1] The F# programming language. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.

[2] The Haskell programming language. www.haskell.org.

[3] M. Abadi, B. Lampson, and J. Lévy. Analysis and caching of dependencies. *ACM SIGPLAN Notices*, 31(6):83–91, 1996.

[4] M. Alvim, K. Chatzikokolakis, P. Degano, and C. Palamidessi. Differential Privacy versus Quantitative Information Flow.

[5] G. Barthe and B. Köpf. Information-theoretic bounds for differentially private mechanisms. Cryptology ePrint Archive, Report 2011/071, 2011. http://eprint.iacr.org/.

[6] C. Dwork. Differential privacy. In *in ICALP*, pages 1–12. Springer, 2006.

[7] C. Dwork, F. Mcsherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *In Proceedings of the 3rd Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[8] B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF*, pages 3–14, 2010.

[9] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI 2008, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 193–205, Tucson, AZ, USA, June 9–11, 2008.

[10] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 19–30. ACM, 2009.

[11] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, Sept. 2010.

[12] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: security and privacy for mapreduce. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[13] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.