Testing Differential Privacy with Dual Interpreters

HENGCHU ZHANG, University of Pennsylvania, USA
EDO ROTH, University of Pennsylvania, USA
ANDREAS HAEBERLEN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
AARON ROTH, University of Pennsylvania, USA

Applying differential privacy at scale requires convenient ways to check that programs computing with sensitive data appropriately preserve privacy. We propose here a fully automated framework for testing differential privacy, adapting a well-known "pointwise" technique from informal proofs of differential privacy. Our framework, called DPCheck, requires no programmer annotations, handles all previously verified or tested algorithms, and is the first fully automated framework to distinguish correct and buggy implementations of PrivTree, a probabilistically terminating algorithm that has not previously been mechanically checked.

We analyze the probability of DPCheck mistakenly accepting a non-private program and prove that, theoretically, the probability of false acceptance can be made exponentially small by suitable choice of test size.

We demonstrate DPCheck’s utility empirically by implementing all benchmark algorithms from prior work on mechanical verification of differential privacy, plus several others and their incorrect variants, and show DPCheck accepts the correct implementations and rejects the incorrect variants.

We also demonstrate how DPCheck can be deployed in a practical workflow to test differentially privacy for the 2020 US Census Disclosure Avoidance System (DAS).

CCS Concepts: • Software and its engineering → Domain specific languages; Software testing and debugging: Dynamic analysis.

Additional Key Words and Phrases: Differential privacy, testing, symbolic execution

ACM Reference Format:
https://doi.org/10.1145/3428233

1 INTRODUCTION

Differential privacy, the highest standard for privacy-preserving data analysis, is seeing increasing real-world deployment [Apple 2017; Microsoft 2017; N. Dajani et al. 2017, etc.]. Differential privacy offers precise guarantees, is robust to arbitrary post-processing, and gives a quantitative estimate of privacy loss. However, differentially private algorithms often require subtle reasoning for their proofs of privacy. Even experts get these proofs wrong [Lyu et al. 2017].
A number of recent efforts have focused on partly or fully automating the process of certifying differential privacy for sophisticated algorithms such as the sparse vector technique [Dwork and Roth 2014]. For example, Zhang and Kifer’s LightDP [Zhang and Kifer 2017a] uses a lightweight dependent type system along with some programmer annotations; Ding et al. [2018] propose a fully automated statistical testing framework that attempts to disprove differential privacy of queries using statistical evidence; Albarghouthi and Hsu [2017] demonstrate a completely automated proof synthesis system with a specialized program logic; and Wang et al. [2019] use a proof technique called “Shadow Execution” to improve upon LightDP and further reduce the amount of programmer annotations required and increase the performance of verification.

We propose a novel framework, called DPCheck, that requires no annotations, handles key benchmark algorithms from previous work, and moreover can accept correct implementation, and reject faulty variants of the PrivTree [Zhang et al. 2016] algorithm, a challenging algorithm for automatic differential privacy verification acknowledged by authors of LightDP [Zhang and Kifer 2017b]. PrivTree was beyond the scope of previous work, owing to a probabilistic main loop that terminates eventually with probability 1 but is not guaranteed to terminate in any bounded number of iterations. Our key insight is that we can combine information from instrumented and symbolic executions of a program to construct privacy proofs for specific executions, then combine these proofs from a large number of executions to give a statistical guarantee of random differential privacy [Hall et al. 2013]. The symbolic interpreter uses information gathered by the instrumented interpreter to build a simple static analysis of the program’s privacy properties, making automated detection of privacy leaks, even for challenging algorithms, feasible.

Following a short review of differential privacy in Section 2 and an overview of DPCheck’s syntax and semantics in Section 3 and Section 4, we offer these main contributions:

1. We present a testing strategy for differential privacy adapted from the pointwise proof technique (Section 5).
2. We prove that the testing strategy always correctly accepts a class of well-behaved differentially private programs, and prove that, in principle, this strategy’s probability of incorrectly accepting ill-behaved (non-private) programs decreases exponentially as test size increases, for a class of ill-behaved programs defined in the framework of random differential privacy (Section 6). We then give an overview of the testing strategy implementation for DPCheck in Haskell (Section 5.4).
3. We demonstrate the effectiveness of the testing strategy by showing that it can detect non-private variants with common programming mistakes, and published mistakes made by experts on the sophisticated benchmark algorithms, and that it accepts correct implementations of all these algorithms. In particular, DPCheck is the first automated framework that can distinguish correct and incorrect variants of PrivTree (Section 7.1). These benchmark algorithms’ differential privacy proofs cover a wide range of complexity, demonstrating DPCheck can analyze both simple and sophisticated differentially private programs.
4. We present a practical workflow that uses DPCheck to re-implement and test the core differential privacy mechanisms in the Disclosure Avoidance System (DAS) [Petti and Flaxman 2019] designed for 2020 US Census; we also show statistical evidence that our re-implemented core mechanism behaves the same as the unmodified DAS (Section 7.2).
5. We implement DPCheck as an embedded language in Haskell and discuss a type-driven optimization adapted from Torlak and Bodik [2014] to speed up symbolic execution, which improves testing time for some our benchmark algorithms (Section 8).

Section 9 enumerates some limitations, and Sections 10 and 11 discuss related and future work.
2 BACKGROUND

A discrete distribution over values of type \( \tau \) is a function of type \( \tau \mapsto [0, 1] \) mapping each value in \( \tau \) to an associated probability. We write \( \mathcal{D} \) for the set of discrete distributions over values in type \( \tau \). An event \( E \) is a subset of \( \tau \). The support of a discrete distribution \( \mu \) is the subset of \( \tau \) whose values have non-zero probability: \( \text{supp}(\mu) = \{ x \in \tau \mid \mu(x) > 0 \} \).

**Definition 1.** Let \( \mu :: \mathcal{D} \) be a discrete distribution. We call \( \mu \) a sub-distribution if the sum of probabilities over its support is at most 1:

\[
\sum_{v \in \text{supp} \mu} \mu(v) \leq 1.
\]

Sub-distributions are useful for describing the semantics of randomized programs because they naturally model non-termination through the “missing probability.” In what follows, we write just “distribution” to mean sub-distribution.

Differential privacy is a relational property of randomized programs. Informally, a program is differentially private if it produces similar distributions when run on similar inputs. The exact similarity relation on inputs depends on what private information we care about protecting. For example, a program \( f \) may be counting the number of patients diagnosed with some disease in a medical database; to conform to regulations, we must not leak the diagnosis of any particular patient. More precisely, the distribution of outputs should be nearly the same if the diagnosis of any single patient changes in the input database. For this example, an appropriate similarity relation on inputs is “at most one patient’s data may be different between the two input databases,” or, more generally:

**Definition 2.** Two multisets have database distance \( k \) if at most \( k \) items must be added or removed to make the two contain exactly the same items.

Another common similarity relation is the \( L_1 \)-distance between two vectors of numbers.

**Definition 3.** The \( L_1 \) distance between vectors \( x_1 \) and \( x_2 \) is the sum of the coordinate-wise distances between corresponding elements of the two vectors.

Finally, some algorithms’ notion of similar inputs is vectors with bounded coordinate-wise distance.

**Definition 4.** Vectors \( x_1 \) and \( x_2 \) have coordinate-wise distance \( k \) if \( |x_1[i] - x_2[i]| \) is bounded by \( k \) for each coordinate \( i \).

Here is the fundamental definition of differential privacy:

**Definition 5.** A randomized program \( f :: \tau \mapsto \mathcal{D} \) is \((\epsilon, \delta)\)-differentially private if, for all similar inputs \( (x_1, x_2) \in \tau \times \tau \), the probability of any event \( E \subseteq \sigma \) satisfies the inequality

\[
\mathbb{P}_{f(x_1)}[E] \leq e^\epsilon \mathbb{P}_{f(x_2)}[E] + \delta.
\]

If the support of the probability distributions is countable, we can simplify the definition of \((\epsilon, 0)\)-differential privacy using a pointwise inequality on the probability difference:

**Definition 6.** A randomized program \( f :: \tau \mapsto \mathcal{D} \), for some countable domain \( \sigma \), is \((\epsilon, 0)\)-differentially private if, for all similar inputs \( (x_1, x_2) \in \tau \times \tau \), the probability of any singleton event \( v \in \sigma \) satisfies

\[
\mathbb{P}_{f(x_1)}[\{v\}] \leq e^\epsilon \mathbb{P}_{f(x_2)}[\{v\}].
\]

The \( \epsilon \) and \( \delta \) in the definition of differential privacy are “privacy parameters.” We can interpret them as a quantitative measure of how much privacy is lost when a sample is observed from the output distribution. As \( \epsilon \) increases, the multiplicative bound on the difference in probabilities of
output events becomes looser, increasing an attacker’s confidence in distinguishing two executions of \( f \) on similar inputs. Experts recommend picking small \( \epsilon \) values (e.g., 1.0) for meaningful privacy protection [Hsu et al. 2014]. On the other hand, \( \delta \) bounds the probability of “catastrophic failure”—failure to provide any privacy at all. It should generally be very small.

DPCheck only guarantees to accept programs that achieve \((\epsilon, 0)\)-differential privacy. However, nonzero \( \delta \)s will play a role in our analysis of false negatives—tests in which DPCheck fails to detect a non-\((\epsilon, 0)\)-differentially private program.

An important tool for writing differentially private algorithms is the Laplace distribution. It is commonly defined as a continuous distribution, but rigorous proofs of differential privacy using continuous distributions require sophisticated measure theory [Sato et al. 2019]. To simplify the required foundations, we follow previous work on program semantics and differential privacy [Albarghouthi and Hsu 2017; Hsu 2017; Reed and Pierce 2010; Wang et al. 2019; Zhang and Kifer 2017b] and assume a discretized, countable support over the reals for all representable numbers. We write \( \omega \) for the constant gap\(^1\) between consecutive values—the granularity of the discretized domain. In this work, we assume all real values are drawn from this discretized domain with granularity \( \omega \).

The discretized Laplace distribution is formally a two-sided geometric distribution [Ghosh et al. 2009]. It is parameterized by a center \( c \) and a parameter \( \alpha \in [0, 1] \). Ghosh et al. [2009] show the two-sided geometric distribution shares the important privacy properties of the continuous Laplace distribution. The continuous Laplace distribution is parameterized by a center \( c \) and a width parameter \( w \) that controls how centered the distribution is around \( c \). Ghosh et al. [2009] also show a straightforward translation between the two-sided geometric distribution parameter \( \alpha \) and the corresponding parameter \( w \) for an equivalent discretized Laplace distribution. We will exclusively use the width parameter \( w \) to parameterize discrete Laplace distributions in this work.

Each rectangle’s area in the graph to the right represents the probability assigned to the point at the center of the base of the rectangle; each rectangle has width exactly \( \omega \)—the granularity of the set of representable numbers. We write \( \text{Lap}_{c, w} \) for the discretized Laplace distribution with center \( c \) and noise width \( w \).

## 3 TYPES AND SYNTAX

DPCheck is a testing framework built around a probabilistic programming language embedded in the functional language Haskell. We will refer to the embedded language itself as DPCheck and to the rest of our system as the “testing framework.”

The DPCheck language offers a simple, purely functional, notation for differentially private programming. DPCheck provides base types `Bool`, `Int`, and `Double`, as well as three container types: tuples \((\tau, \sigma)\), lists \([\tau]\), and maps \(\text{Map} \tau \sigma\). Finally, DPCheck supports probabilistic programming through a distribution monad \(\circ\): the type \(\tau \circ \tau\) represents a (sub-)distribution over values of type \(\tau\). Unlike a number of previously proposed functional languages for differential privacy [Gaboardi et al. 2013; Reed and Pierce 2010, etc.], DPCheck’s type system does not track privacy: this is the job of the testing framework.

We embed DPCheck inside Haskell, using methods developed by Svenningsson and Axellson [2013, 2015] for the Feldspar language [Axelsson et al. 2010]. Their key insight is a technique for combining “deep” and “shallow” representations of programs, where a deep representation for a program is an abstract syntax tree, while a shallow representation maps language constructs directly to their semantics. Here, the deeply represented parts of the language can be used by

---

\(^1\)We present DPCheck and its properties using this idealized set of representable reals, but our implementation relies on floating point numbers. This discrepancy and its impact on testing are discussed in Section 9.
DPCheck’s symbolic interpreter for static analysis, while the shallowly represented parts save engineering on things like surface syntax, standard libraries, and compilation by borrowing from the host language.

DPCheck’s deep representation in Haskell uses values of an indexed datatype `Expr`. For example, a Haskell term of type `Expr Bool` represents a DPCheck program that yields values of type `Bool` when evaluated. The type index (`Bool`) allows us to borrow Haskell’s typechecker to rule out ill-formed programs statically.

DPCheck’s most important syntactic forms are:

```haskell
data Expr a where
  Literal :: a -> Expr a
  Add    :: Expr Double -> Expr Double -> Expr Double
  Lt     :: Expr Double -> Expr Double -> Expr Bool
  If     :: Expr Bool -> Expr (a) -> Expr (a) -> Expr (a)
  Laplace :: Expr Double -> Double -> Expr (a)
  Return :: Expr a -> Expr (a)
  Bind   :: Expr (a) -> (Expr a -> Expr (b)) -> Expr (b)
```

These are Haskell constructors allowing us to build DPCheck programs that perform arithmetic using `Add`, compare numeric values using `Lt`, branch on boolean conditions using `If`, sample from Laplace distributions using `Laplace`, sequence probabilistic computations using `Bind`, and create point mass distributions using `Return`. The `Bind` and `Return` constructors endow DPCheck with monadic structure [Moggi 1989], allowing computations that return distributions to be coded in a natural style. Additionally, the purity of Haskell automatically rules out programs with potentially non-private side-effects (such as using the current system time to calculate an argument to `Literal`), because the types of the constructors only allow effect manipulation related to probability distributions.

As an example, we consider implementing the ReportNoisyMax algorithm in DPCheck. The ReportNoisyMax algorithm returns the index of the largest noisy value. Assuming an input list of `[1, 2, 10]` to ReportNoisyMax, and the noise values are `0.8`, `-1.2` and `-0.9`, ReportNoisyMax returns the index of the largest value from the noised list `[1.8, 0.8, 9.1]`. We can define ReportNoisyMax with the following Haskell syntax and a Haskell library combinator `mapM` that applies a function uniformly over a list:

```haskell
rnm :: [Expr Double] -> Expr (Int)
rnm (x:xs) = do
  xNoised ← lap x 1.0
  xsNoised ← mapM (λy. lap y 1.0) xs
  rnmAux xsNoised 0 0 xNoised
rnm [] = error " rnm received empty input"

rnmAux :: [Expr Double] -> Expr Int -> Expr Int -> Expr Double
  ... (Expr a -> Expr (a) "(Expr a -> Expr (a) "
```

---

2 The width parameter to the `Laplace` constructor has type `Double` instead of `Expr Double`. This implies the width parameter must be a statically chosen constant. This restriction simplifies the testing framework implementation, and it does not rule out any algorithms in our evaluation.

3 In the current implementation, it is still possible to escape `Expr`’s restrictions by using unsafe language features to subvert Haskell’s type system. We could fix this by requiring DPCheck programmers to only use Safe Haskell [Terei et al. 2012], ruling out such subversions completely. In this paper, we assume the programmer is not adversarial and only wants to test programs that she genuinely believes are differentially private.
The left pointing arrow ← is a syntactic sugar of the bind constructor for sequencing probabilistic computations. Intuitively, the syntax x ← m represents a program fragment that runs the probabilistic computation m, giving the result of that computation a name x, and allows x to be used in following computations.

Conceptually, executing DPCheck programs through Haskell is a two-stage process: Haskell itself becomes a host language for constructing embedded DPCheck programs, which are then executed using an eval function (see the extended version). This arrangement allows programmers to use convenient Haskell syntax and libraries when writing DPCheck programs.

An unusual aspect of DPCheck’s syntax design is that we rely on recursion in Haskell for—in effect—generating iterative DPCheck code. This design relies critically on Haskell’s support for lazy evaluation. Iteration through host-level recursion, if implemented in a strictly evaluated host language, would restrict the kinds of algorithms that can be represented, since a strict host-language would fully construct the syntax tree before sending it off to an interpreter. This implies all loops built with strict host-language recursion would be fully unrolled immediately. Some programs may have unbounded loops in some control flow paths, and such programs will cause divergence under this scheme. Fortunately, since Haskell is lazily evaluated, its runtime does not construct a value until the value’s contents are required. This means the Haskell code that constructs DPCheck syntax does not run until that syntax is required by the interpreter. In our implementation, we rely on Haskell’s lazy evaluation to automatically interleave DPCheck code execution and (potentially infinite) loop unrolling.4

When the function rnm is applied to an input list xs, we obtain a closed DPCheck program with type Expr ○ Int. This program contains data that represents DPCheck commands to be interpreted. The initial commands are Laplace sampling commands that add noise to the input list. These commands are generated with $\lambda$ y. lap y 1.0, which builds the Laplace nodes in a DPCheck program’s syntax tree, and mapM, which applies this function for all items in a list. We then loop over the noised data, keeping track of the maximum value and its index seen so far through if commands, and return the index at the end with the helper function rnmAux.

4 SEMANTICS

We can straightforwardly encode DPCheck’s evaluation semantics using a function eval with type Expr r → r (see the extended version for details). Recall that a differentially private program whose syntax tree has type Expr ○ a is necessarily probabilistic. Calling eval on such a program results in a distribution (value) of type ○ a, and we can sample from this distribution to acquire a concrete output value.

Besides this evaluation semantics, the testing framework also uses symbolic execution [King 1976] of DPCheck programs. During symbolic execution, concrete samples from Laplace distributions are replaced by symbols, allowing us to explore all possible control flows throughout a DPCheck program, even when branch conditions depend on sampled values. The symbolic execution process records the branch conditions along each control flow path; we call these recorded boolean conditions path conditions. As an example, if we run rnm on an input list of length two, then the symbolic execution process will create two symbolic values $s_0$ and $s_1$, one for each noised sample value. On one path, we explore under the assumption that the branch condition $s_0 > s_1$ evaluates to True, and this results, say, in the output value 0; on the other path, we explore under

---

4 Although the ordinary DPCheck interpreter handles infinite syntax trees thanks to laziness, this doesn’t help the symbolic interpreter, because it tries to explore all paths. However, we can use auxiliary information to stop potentially infinite symbolic execution as soon as no further control flow path exploration is necessary. See Section 9.
the assumption that $s_0 > s_1$ evaluates to False, and this results in the output value 1. So, in this example, symbolically executing rnm on this input list of length two yields two possible output values 0 and 1, with path conditions $s_0 > s_1 = True$ and $s_0 > s_1 = False$.

5 TESTING DIFFERENTIAL PRIVACY

To show how DPCheck tests differential privacy, we first review how proofs of differential privacy are constructed (5.1), with ReportNoisyMax as an example (5.2). Then we discuss how to adapt the same basic ideas to testing (5.3).

5.1 Background: apRHL

Differential privacy experts often approach proofs in a “pointwise” fashion using Definition 6: Given two executions on similar inputs, first demonstrate that, no matter what concrete value the first execution of rnm yields, the second execution can also produce the same result; then show that the multiplicative difference in the total probability of all executions that lead to these identical outputs is bounded by the algorithm’s prescribed privacy parameter.

Barthe et al. [2016a] formalized the pointwise proof technique in a program logic for differential privacy called apRHL (approximate, relational Hoare logic). A key innovation of apRHL is its notion of “approximate lifting,” which abstracts over relations between distributions. Approximate lifting allows us to use a deterministic relation to simultaneously couple all samples from one distribution with samples from the other distribution, effectively reducing probabilistic reasoning to deterministic reasoning. The details of apRHL are beyond the scope of this paper (Hsu 2017) is a readable introduction); here we just sketch the parts on which our testing strategy is most immediately based.

An apRHL judgment for differential privacy has the form $\vdash c \sim (\epsilon, \delta) : \Phi \Rightarrow \Psi$, where $c$ is a randomized program. It describes a relation between the output distributions on related executions of $c$. Here, $\Phi$ is a pre-condition on the free variables of $c$ between related executions, and $\Psi$ is a deterministic relation on the output samples of $c$ that, behind the scenes, is lifted into the corresponding relation between the output distributions through apRHL’s approximate lifting machinery. The parameters $(\epsilon, \delta)$ represent the “cost” of establishing the post-condition relation $\Psi$. When $\Psi$ asserts that the related outputs of $c$ are equal, which implies differential privacy for the program $c$, the parameters $(\epsilon, \delta)$ are exactly the privacy cost.

For example, to state that rnm is $(2, 0)$-differentially private using an apRHL judgment, we first encode the similarity relation of the inputs in the precondition. Two similar inputs of rnm must have coordinate-wise distance bounded by 1, which we encode with the assertion $\bigwedge_i |xs_1[i] - xs_2[i]| < 1$. Then, to encode differential privacy, we assert the outputs of two executions on similar inputs are identical in the post condition:

$$\vdash (out_1 \leftarrow rnm\ xs_1) \sim (2, 0) (out_2 \leftarrow rnm\ xs_2) : \left(\bigwedge_i |xs_1[i] - xs_2[i]| < 1\right) \Rightarrow (out_1 = out_2).$$

Two key proof rules that formalize the pointwise proof technique are called Lap-Gen and PW-Eq. The Lap-Gen rule allows us to connect Laplace samples in two executions with a deterministic relation through apRHL’s approximate lifting theory. In particular, this deterministic relation allows

5More generally, a relation can be established between syntactically different program fragments $c_1$ and $c_2$ instead of a single program $c$. This general form of apRHL judgment is useful for intermediates steps of proofs; we ignore this refinement here for the sake of simplicity.
us to assume (in the postcondition) that the samples drawn from the Laplace distribution on the two runs are at a fixed distance $k$ apart.

$$\text{Lap-Gen}$$

$$\epsilon = |k + e_1 - e_2|/w$$

$$\vdash (x_1 \leftarrow \text{lap } e_1 w) \sim_{(\epsilon, 0)} (x_2 \leftarrow \text{lap } e_2 w) : \Phi \Rightarrow (x_1 + k = x_2)$$

Readers encountering couplings for the first time may worry that this proof rule looks too good to be true, since it allows us to assume related samples are always at a deterministic distance $k$ apart. What makes this work is that we are not considering some particular pair of samples; rather, we are relating the entire support from two Laplace distributions simultaneously. Furthermore, establishing such a relation in the post-condition does not come for free: we are allowed to choose any $k$, but the privacy cost also depends on $k$.

This deterministic rule abstracts away direct reasoning over the related Laplace distributions. In Section 5.2, we will present an example that applies Lap-Gen to prove $\text{rnm}$ is $(2, 0)$-differentially private. We will refer to these $k$ values used in Lap-Gen rules as “shift values.”

The PW-Eq rule is a formal description of the pointwise proof technique: if we can show that, for each possible output value $r \in \tau$, one execution returning $r$ implies that the other execution also returns $r$, then these pointwise facts together constitute a complete differential privacy proof.

$$\text{PW-Eq}$$

$$e_1, e_2 :: \tau \quad \forall r \in \tau, \quad \vdash c_1 \sim_{(\epsilon, 0)} c_2 : \Phi \Rightarrow (e_1 = r \rightarrow e_2 = r)$$

$$\vdash c_1 \sim_{(\epsilon, 0)} c_2 : \Phi \Rightarrow (e_1 = e_2)$$

The variables $e_1$ and $e_2$ here represent the output values of $c_1$ and $c_2$ respectively. The power of PW-Eq is that, for each $r \in \tau$, we are allowed to choose a different shift value for any application of the Lap-Gen proof rule that may appear in the subderivation for this $r$. To prove $\text{rnm}$ is $(2, 0)$-differentially private, we will apply exactly this strategy: given any output $r$ from one execution of $\text{rnm}$, we choose some sequence of shift values so that the noisy max also occurs at $r$ in the second execution, forcing the second execution to also return $r$.

With these two proof rules, the process of proving differential privacy for ReportNoisyMax reduces to a simpler goal: for each possible output of ReportNoisyMax, find a sequence of shift values to relate the Laplace samples between two executions so that their output values are identical.

Section 5.3 will show how we use the ideas behind the Lap-Gen and PW-Eq rules in designing our testing strategy for differential privacy. But first, a concrete example.

5.2 An Example Privacy Proof

Let’s consider a (paraphrase of a) proof by Dwork and Roth [2014] of $(2, 0)$-differential privacy for $\text{rnm}$ when the coordinate-wise distance (Definition 4) between the inputs is bounded by 1. We present this proof by sketching applications of Lap-Gen and PW-Eq, and emphasizing steps in the proof that will become key ingredients in our testing algorithm by putting boxes around equations. This proof can be carried out formally in $\text{apRHL}$, but we present it informally to avoid getting bogged down in details.

Theorem 7. ReportNoisyMax is $(2, 0)$-differentially private.

---

6This rule may appear to be showing—counterintuitively—that different executions of a randomized program produce identical results. Of course, any two concrete executions of a randomized program will almost certainly produce different results. But we are not actually reasoning here about some particular pair of executions. Rather, we are reasoning simultaneously about all pairs of coupled executions.
PROOF. The implementation of ReportNoisyMax creates a list of noised values based on its input. Let \( r \) be any possible output of running \( \text{rnm} \). Let \( \text{argmax} \) be the function that returns the index of the largest value in a list. When \( \text{rnm} \) runs on an list of input values, it first adds Laplace noise to each of these values, then iterates over the noised values, keeping track of the index of the maximum value seen so far, and finally returns that index. Write \( qs'_1 \) and \( qs'_2 \) for the two intermediate lists of noised values from \( qs_1 \) and \( qs_2 \). Then, if \( r \) is the result of running \( \text{rnm} \) on \( qs_1 \), it is easy to see that \( r = \text{argmax} \, qs'_1 \).

We next apply the pointwise proof technique described in Section 5.1. Since we assumed one execution of \( \text{rnm} \) returned index \( r \), we need to show some control flow through the run of \( \text{rnm} \) on \( qs_2 \) yields the same \( r \). We demonstrate such a control flow by carefully choosing the shift values introduced in the \( \text{Lap-Gen} \) proof rule. Concretely, we need to ensure \( r = \text{argmax} \, qs'_2 \), under some choice of the shift values that connects each \( qs'_1[i] \) and \( qs'_2[i] \).

We proceed by applying \( \text{Lap-Gen} \) to each pair of noised values with the following choice of shift values, coupling each \( qs'_1[i] \) with \( qs'_2[i] \) such that \( qs'_2[i] = qs'_1[i] + \text{shift}_i \):

\[
\text{shift}_i = \begin{cases} 
1 & \text{if } i == r \\
qs_2[i] - qs_1[i] & \text{otherwise}
\end{cases}
\]

(2)

This choice implies that the maximum value in the second noised array also occurs at index \( r \):
we know that \(-1 < qs_2[i] - qs_1[i] < 1 \) for all \( i \), since \( qs_1 \) and \( qs_2 \) are similar inputs. In particular, \( qs_2[i] - qs_1[i] < 1 \). Adding \( qs'_1[i] \) to both sides of this inequality yields

\[qs'_2[i] + (qs_2[i] - qs_1[i]) < qs'_1[i] + 1.\]

Since \(qs'_2[i] = qs'_1[i] + (qs_2[i] - qs_1[i])\) if \( i \neq r \) and \(qs'_2[r] = qs'_1[r] + 1\), it follows that \( r = \text{argmax} \, qs'_2 \).

We have shown so far that, for any output index \( r \) from the first execution, it is possible for \( \text{rnm} \) to produce the same result \( r \) on the similar input. Next we need to calculate the \( \epsilon \) privacy cost incurred by using \( \text{Lap-Gen} \) for connecting \( qs'_1 \) with \( qs'_2 \), and prove \( \epsilon \) is at most 2.

Consider the cost of an application of \( \text{Lap-Gen} \) between a given pair of \( i \)th sample values \( qs'_1[i] \) and \( qs'_2[i] \) in the two runs; call this \( \text{cost}_i \). Under apRHL, the total privacy cost is bounded by the sum of the \( \text{cost}_i \)'s, so we need to show \( \sum \text{cost}_i < 2 \).

To bound the sum, let us first consider the cost values for indices \( i \neq r \). We know \( qs'_2[i] \) is a sample from the distribution \( 1\text{ap}_{qs_2[i],1.0} \), and similarly \( qs'_1[i] \) is a sample from \( 1\text{ap}_{qs_1[i],1.0} \). Using \( \text{Lap-Gen} \), we can conclude

\[\text{cost}_i = |(qs'_1[i] + qs_2[i] - qs_1[i]) - (qs'_2[i] - qs_1[i])|\]

\[= 0.\]

On the other hand, if \( i = r \), let \( s = qs'_1[r] - qs_1[r] \). We again use \( \text{Lap-Gen} \) to conclude:

\[\text{cost}_r = |(qs'_1[r] - qs_1[r]) - (qs'_2[r] - qs_2[r])|\]

\[= |s - (qs'_1[r] + 1 - qs_2[r])| \quad \text{by eq. (2) and assumption of } s\]

\[= |s - (qs_1[r] + s + 1 - qs_2[r])| \quad \text{by assumption of } s\]

\[= 1|qs_1[r] - qs_2[r]| + 1|\]

\[< 1 + 1 = 2 \quad \text{by the triangle inequality}\]

(3)

Since cost values are 0 for indices \( i \neq r \) and less than 2 when \( i = r \), the total cost is less than 2. That is, \( \text{rnm} \) is \((2, 0)\)-differentially private. \(\Box\)
5.3 From Proving to Testing

The key steps in the above proof are:

1. Assume an unknown but fixed output \( r \) from one execution.
2. Select a sequence of shift values (eq. (2)) to connect the samples from one execution with samples from the other execution.
3. Show that the second execution, whose Laplace samples are fixed through the shift values, leads to the same output \( r \).
4. Compute the total privacy cost of this pair of executions as the sum of the individual cost values induced by the chosen sequence of shift values and show this total is less than the prescribed \( \epsilon = 2 \) (eq. (3)).

In order to convert this to a testing procedure, we need to check that, for a run of \( rnm \) on \( qs_1 \), there exists a dual execution of \( rnm \) on \( qs_2 \) leading to the same output, such that the Laplace samples in these two executions can be connected through a sequence of shift values, while keeping the total privacy cost induced by the shift values under 2.0. However, we need to be careful about how many distinct sequences of shift values are allowed for constructing dual executions. The PW-Eq rule allows us to choose one sequence of shift values for each unique output \( r \) that \( rnm \) may return. Thus, for testing, we must group the runs of \( rnm \) on \( qs_1 \) by their output, and then try to find a single sequence of shift values for each group, such that this single sequence of shift values leads to corresponding executions of \( rnm \) on \( qs_2 \) with the same output.

With this in mind, the first step in testing is to run \( rnm \) on \( qs_1 \) a large number of times and group the runs by their final output values.

Next we need to find the shift values. In steps 2 and 3 of the proof above, we used expert insight to select shift values that allowed us to show that the dual execution must result in the same output \( r \). When testing, we begin with hypothetical (symbolic) shift values and hope to work out their concrete values later: we create one symbolic shift value for each \( qs_1'[i] \) per group, and pair \( qs_1'[i] \) with a \( qs_2'[i] \) through the following equation

\[
qs_2'[i] = qs_1'[i] + shift_i. \tag{4}
\]

This equation is exactly how Lap-Gen allows us to connect two Laplace samples. (Throughout this discussion, we highlight the important symbolic formulas created in the testing steps by boxing them.) This also induces a symbolic cost using Lap-Gen, where

\[
cost_i = \frac{|(qs_2'[i] - qs_2[i]) - (qs_1'[i] - qs_1[i])|}{w} = \frac{|qs_1[i] + shift_i - qs_2[i]|}{w}. \tag{5}
\]

Here \( qs_1[i] \) is the concrete center supplied to the \( i \)th call to the sampling instruction in the first execution, \( shift_i \) is a fresh symbolic variable, \( qs_2[i] \) is the concrete center used in the \( i \)th sampling instruction in the second execution, and \( w \) is the (known constant) parameter controlling the width of the Laplace distributions used in \( rnm \). Thus, both \( qs_2'[i] \) and \( cost_i \) can be represented as symbolic expressions coupled with concrete execution traces using the formulas above if we know what \( qs_1[i] \) and \( qs_1'[i] \) are.

To capture \( qs_1 \) and gather many independent samples of \( qs_1' \) for testing, we repeatedly run \( rnm \) on \( qs_1 \) with a special interpreter that instruments Laplace sample instructions, recording the center, width, and returned sample value for each. We group each unique output \( r \) together with all sequences of Laplace sample and parameters that lead to the output \( r \) into a bucket.
For each bucket $bkt$, we need to show there exists a coupling that connects all sampled traces in $bkt$ to the Laplace samples from running on $qs_2$, such that this coupling leads the dual execution on $qs_2$ to the same result $r$ from $bkt$. To find such a coupling, we symbolically execute $rnm$ with symbolic samples $qs'_2$, observing all possible outputs of $rnm$ along all of the control flow paths. Among these paths, some will return $r$. We gather all of the path conditions from control flow paths that lead to $r$. These path conditions are then used to constrain the shift values from eq. (4), so that the coupled dual execution only takes the paths that yield the same output $r$. We use $\Phi_r$ to denote the symbolic formula that encodes the path conditions, and $\Omega_r$ to denote the symbolic formula that encodes the shift equations (eq. (4)). These two symbolic formulas form the testing analog of step 2 and 3 from the proof.

Finally, to bound the total privacy cost (as in step 4 of the proof), we create another symbolic formula using the symbolic expressions for each $cost_i$:

$$\sum_i cost_i \leq 2$$

Together, the boxed symbolic formulas constitute a query that can be dispatched to an off-the-shelf SMT solver—we use Z3 [De Moura and Bjørner 2008]. If the solver returns a satisfying model for these constraints, then we know the distributions produced by running $rnm$ on this particular pair of $qs_1$ and $qs_2$ likely satisfy 2-differential privacy. Of course, this does not guarantee differential privacy, because we used sampled traces of Laplace calls to describe properties of a potential proof for differential privacy, instead of universally quantifying over all possible samples in one execution, as in the proof.

At the core of both the proof and the SMT formula is the relation between $qs'_1$ and $qs'_2$. The proof demonstrates that there exist shift values such that the related Laplace samples satisfy the privacy cost bounds. The testing process also checks for the existence of such shift values; however, it has a chance of admitting programs that are not $(\epsilon, 0)$-differentially private, because testing does not produce a complete proof of differential privacy—it only checks whether dual executions satisfying differential privacy exist on some set of sampled traces. There are two important test parameters—number of pairs of randomly generated similar inputs, and number of sampled traces collected on a given pair of similar input—that can be independently tuned to make tests more difficult to pass for faulty programs. Intuitively, as we increase both test parameters, it should be less and less likely that our testing process accepts a faulty program. We call programs that are faulty but slip past DPCheck’s testing framework false negatives.

A non-$(\epsilon, 0)$-differentially private program may be faulty for two reasons: 1) it may have a non-zero $\delta$ failure probability, and 2) there may exist similar inputs $(x_1, x_2)$ for which the program produces distributions that do not satisfy definition 6 with the given $\epsilon$. Note that the definitions of differential privacy (5 and 6) require the relation on output distributions in definition 6 to hold on all similar inputs. Although DPCheck’s testing framework can check that eq. (1) holds on a pair of fixed similar inputs by checking the existence of shift values and dual executions, it can never exhaustively check that eq. (1) holds on a potentially infinite set of similar pairs of inputs.

Instead, DPCheck gives probabilistic guarantees using the framework of random differential privacy [Hall et al. 2013]. In the next section, we review random differential privacy and use it to state our main theoretical testing guarantee (Theorem 12).

### 5.4 Implementation

We now describe the testing process more concretely, showing type signatures of Haskell functions that implement key testing steps.
DPCheck’s testing framework takes as inputs a program under test, \( \text{prog} :: \sigma \rightarrow \text{Expr} (\bigcirc \ r) \), a generator, \( \text{gen} :: \text{Gen} (\sigma, r) \), and a privacy parameter \( \epsilon \). It then checks, for a large number of \((x_1, x_2)\) pairs produced by \( \text{gen} \), that the distributions produced by running \( \text{prog} \ x_1 \) and \( \text{prog} \ x_2 \) satisfy eq. (1). If it ever finds one that does not, it rejects \( \text{prog} \); otherwise it validates \( \text{prog} \) as likely to be \((\epsilon, 0)\)-differentially private. (We discuss what “likely” means more formally in Section 6.) We also show how much time DPCheck takes to reject buggy benchmark programs in the extended version.) The more tests a program passes, the more likely the program really is \((\epsilon, 0)\)-differentially private. Our experiments on benchmark algorithms show that DPCheck rejects many faulty programs within 10 seconds, but it may take significantly longer testing time to reject algorithms that only demonstrate privacy violations on larger inputs (see the extended version for detailed benchmark study).

To verify that the distributions produced by a particular pair of similar inputs are related, we need to construct a coupling between the Laplace distribution samples used by \( \text{prog} \ x_1 \) and \( \text{prog} \ x_2 \) using the methods described in Section 5. We first acquire concrete sample values from some large number (call it \( N \)) of runs of \( \text{prog} \ x_1 \). We denote the output from the \( i \)th run by \( r_i :: \tau \) and the sample trace from each of these runs by \( tr_i :: \text{Trace} \), where

\[
\text{type SampleInfo} = (\text{Double}, \text{Double}, \text{Double})
\]

and where the following projections extract sample, center, and width values from a \( \text{SampleInfo} \):

\[
\text{sample}, \text{center}, \text{width} :: \text{SampleInfo} \rightarrow \text{Double}
\]

With the collected outputs \( r_1, r_2, \ldots, r_N \) and traces \( tr_1, tr_2, \ldots, tr_N \), we perform a “bucketing” process so that all traces that lead to the same output value \( r \) are grouped together.

\[
\text{type Buckets} \ r = \text{Map} \ r \ [\text{Trace}]
\]

bucket :: [(r, Trace)] \rightarrow \text{Buckets} \ r

A value of this map type represents a collection of buckets; a particular key-value pair (of an output value with its associated list of traces) is a single bucket.

Next, we perform symbolic execution on \( \text{prog} \ x_2 \). To do this, we first perform a simple program transformation: \( \text{streamline} :: \text{Expr} (\bigcirc \ r) \rightarrow [\text{Expr} (\bigcirc \ r)] \). This transformation repeatedly replaces a program containing \( \text{if} \) commands with two new programs in which the \( \text{if} \) command is replaced by sequencing \( \text{Assert} \ \text{cond} \) with the commands in the true branch and sequencing \( \text{Assert} \ \text{not} \ \text{cond} \) with commands in the false branch.

\[
data \text{Expr} \ a \ where
\]

\[
\begin{align*}
\text{Assert} :: & \text{Expr} \ \text{Bool} \rightarrow \text{Expr} \ (\bigcirc \ ()) \\
\text{Sequence} :: & \text{Expr} \ (\bigcirc \ ()) \rightarrow \text{Expr} \ (\bigcirc \ a) \rightarrow \text{Expr} \ (\bigcirc \ a)
\end{align*}
\]

This simplistic approach produces \( 2^n \) straight-line programs in the worst case, where \( n \) is the number of \( \text{if} \) statements; we will discuss a type-driven optimization adapted from \cite{Torlak and Bodik 2014} for speeding up symbolic execution in Section 8.

Note that \( \text{streamline} \) would actually diverge on infinite syntax trees if Haskell were not lazy. Our symbolic interpreter uses information gathered by instrumented implementation to cut off infinite symbolic execution as soon as possible. We use this early cutoff trick in the evaluation of \( \text{PrivTree} \); the trick and some directions for generalizing it are discussed in more detail in Section 9.

\footnote{A value of type \( \text{Gen} \ r \) is a function that takes a seed and produces a pseudo-random value of type \( r \).
}

\footnote{For invalidating incorrect algorithms, we can start with small \( N \) (such as 50) and keep increasing \( N \) by a factor of 10 until the bug is discovered. For validating correct algorithms, \( N \) should be chosen according to the lower bound on \( m \) in Theorem 12. However, currently the computational cost of running tests with large \( N \) makes validation prohibitively slow. We discuss this issue in Section 9.
}
The programs resulting from this transformation are free of conditional branches, instead explicitly encoding path conditions using Assert nodes. We next take these transformed programs and run symbolic execution guided by the trace buckets from the instrumented executions above.

Consider a particular set of executions that lead to the output value \( r \), and let the associated trace bucket contain the traces \( tr_{i_1}, tr_{i_2}, \ldots, tr_{i_k} \), where \( i_1, i_2, \ldots, i_k \) are the indices of instrumented runs that produced \( r \). We then search for paths that produce the same output \( r \) and build eq. (4) between the concrete sampled traces and the symbolic Laplace samples. For each trace \( tr_{i_k} \), we pair it with symbolic Laplace samples as follows: on the \( j \)th call to the Laplace sampling instruction during symbolic execution, we create a fresh symbolic value \( lap_{i_k}[j] = \text{sample}(tr_{i_k}[j]) + \text{shift}_j \). Let \( \psi_k = \bigwedge_j lap_{i_k}[j] = \text{sample}(tr_{i_k}[j]) + \text{shift}_j \), and \( \Psi_r = \bigwedge_k \psi_k \), and let \( \Phi_r \) encode the disjunction of the path conditions for all control flow paths that lead to the output \( r \).

The final formula \( \Psi_r \land \Phi_r \land (\sum_n \text{cost}_n \leq \epsilon) \) asserts that \( \text{prog} \) produces the same output \( r \) within the prescribed privacy cost \( \epsilon \). We perform the same process for each unique output \( r \) observed from instrumented executions. If these formulas are all satisfiable, we consider this test a passing test case, and we do not reject the claim of \((\epsilon, 0)\)-differential privacy. On the other hand, if, for some output \( r \), Z3 tells us that the formula is not satisfiable, then the program under test does not have a point-wise proof of \( \epsilon \)-differential privacy using the proof template discussed in Section 5. This is not a disproof of differential privacy, since the proof template we are using is not complete (there are algorithms whose privacy proofs do not follow this pattern), but it is at least a signal that should prompt us to look at the program skeptically.

The code below sketches the testing process on an input procedure to test \( \text{prog} \), a pair of neighboring inputs \( x_1 \) and \( x_2 \), an expected privacy parameter \( \epsilon \), and the number of sampled traces to draw for testing \( ntraces \).

```plaintext
expectDP \text{prog} \text{x1} \text{x2} \epsilon \text{ntraces} = \text{do}
  \text{buckets} \leftarrow \text{instrumentedExec ntraces (prog x1)}
  \text{let} \text{straightlineProgs} = \text{streamline (prog x2)}
  \text{constraints} \leftarrow \text{symbolicExec buckets straightlineProgs}
  \text{solverResults} \leftarrow \text{runSolver eps constraints}
  \text{expect} (\text{all isOk solverResults})
```

A distinctive element of our design is the combination of instrumented execution and symbolic execution. An alternative would be to run symbolic execution on both inputs using relational symbolic execution [Farina et al. 2017], then universally quantify over the Laplace samples in one execution, as demonstrated in the analysis of \( \text{rm} \) from Section 5. Using relational symbolic execution would bring more confidence to the differential privacy property of the program under test, since the satisfying model from the SMT solver will serve as a formal proof of eq. (1) for the pair of output distributions. However, this approach produces more complex symbolic formulas that may significantly slow down Z3. To strike a balance between execution time and confidence gained from a passing test, we choose to combine instrumented execution and symbolic execution.

### 6 ASYMPTOTIC PRIVACY GUARANTEES

In this section, we introduce random differential privacy (RDP), and use RDP to quantitatively define “well-behaved” and “ill-behaved” programs. We present Theorem 12, which gives an upper bound on the probability of DPCheck falsely accepting ill-behaved programs. However, DPCheck has a scaling bottleneck that prevents us from applying Theorem 12 to produce meaningful guarantees of random differential privacy on correct programs. We discuss details of the scaling bottleneck and its implication on statistical validation of correct programs in Section 9.

To introduce random differential privacy, let us first consider an alternative view of the privacy parameters \( \epsilon \) and \( \delta \).

Definition 8. Let $\mu_1, \mu_2 : \tau \mapsto \mathbb{R}$. Let $f(\tau) = \ln \frac{\mu_1(\tau)}{\mu_2(\tau)}$. The privacy loss random variable is a distribution $\text{pv}(\mu_1, \mu_2) \in \bigodot \mathbb{R}$ defined as: $\text{pv}(\mu_1, \mu_2)(x) = \sum_{\tau \in \text{supp}(\mu_1)} \mathbb{I} f(\tau) = x \mu_1(\tau)$.

Informally, this distribution can also be expressed as $f(\tau)$, where the random variable $\nu \sim \mu_1$.

Definition 9 ([Kasiviswanathan and Smith 2014]). Two distributions $\mu_1$ and $\mu_2$ are $(\epsilon, \delta)$-pointwise indistinguishable if the probability mass of $\text{pv}(\mu_1, \mu_2)$ in the interval $[-\epsilon, \epsilon]$ is at least $1 - \delta$.

Note that $(\epsilon, \delta)$-pointwise indistinguishability implies eq. (1) in the definition of differential privacy (Definition 5). Furthermore, the proof template we introduced in Section 5 constructs proofs of $(0,0)$-pointwise indistinguishability.

Definition 10 ([Hall et al. 2013]). Assume a fixed distribution over similar inputs $I : \bigodot (\sigma \times \sigma)$. A randomized program $f : \sigma \mapsto \tau$ is $(\epsilon, \delta, \alpha)$-random differentially private if, with probability at least $1 - \alpha$, sampling similar inputs $(x_1, x_2)$ from $I$ leads to $(\epsilon, \delta)$-pointwise indistinguishable distributions $f(x_1)$ and $f(x_2)$.

We can give some intuition for Definition 10 by considering visual graphs of the privacy loss random variable under given similar inputs. First, assume a distribution $I$ of similar inputs. Let us sample nine pairs of similar inputs from $I$ and draw their privacy loss random variables as a graph centered at 0 on the horizontal axis. We shade each graph with blue if its area is at least $1 - \delta$ in the interval $[-\epsilon, \epsilon]$, and red otherwise.

Under Definition 9, the two distributions are $(\epsilon, \delta)$-pointwise indistinguishable if the shaded area in the interval $[-\epsilon, \epsilon]$ is at least $1 - \delta$. Using this visual criterion, the definition of $(\epsilon, \delta, \alpha)$-differential privacy says that, if we repeatedly sample similar inputs from $I$ and inspect the corresponding graph of the privacy loss random variable, then with probability at least $1 - \alpha$, we will see a graph whose shaded area is at least $1 - \delta$. The example graphs show seven privacy loss random variable distributions whose shaded area in the interval $[-\epsilon, \epsilon]$ is at least $1 - \delta$ (colored in blue), and two whose shaded area is less than $1 - \delta$ (colored in red). Visually, if we extend this grid of graphs with privacy loss random variables derived from more and more sampled similar inputs from $I$, then the parameter $\alpha$ bounds the fraction of red graphs in the entire grid.

We say a program $f$ is $\epsilon$-well-behaved if $f$ is $(\epsilon, 0)$-differentially private and if $f$’s path conditions are exactly necessary and sufficient for proving $(\epsilon, 0)$-differential privacy through the pointwise proof technique.

Lemma 11. If a program $f$ is $\epsilon$-well-behaved, then $f$ is never rejected by DPCheck’s testing framework when tested with any $\epsilon' \geq \epsilon$.

The proof of Lemma 11 can be found in the extended version.

We might hope that all $(\epsilon, 0)$-differentially private programs are $\epsilon$-well-behaved, but this does not hold in general because DPCheck assumes that proofs of differential privacy for these programs have a particular structure (Section 5); the path conditions for these programs must be the necessary and sufficient conditions for its privacy properties. In Section 7 we will see the ReportNoisyMaxWithGap algorithm, whose optimal $\epsilon$ is rejected by DPCheck because its path conditions are sufficient but not necessary. (DPCheck does accept ReportNoisyMaxWithGap with a non-optimal $\epsilon$, for which its path conditions are both necessary and sufficient.)
Conversely, assume a fixed distribution $I$ of similar inputs and a program $f$. We say that $f$ is $(\epsilon, \delta, \alpha)$-ill-behaved if, given fixed $\epsilon$, all valid random differential privacy parameters $(\epsilon, \delta_f, \alpha_f)$ for $f$ satisfy $\delta_f > \delta$ and $\alpha_f > \alpha$. Intuitively, an $(\epsilon, \delta, \alpha)$-ill-behaved program has a high probability of two kinds of failure—its “catastrophic failure” probability is at least $\delta$ when executed on “good” similar inputs from $I$, and there is at least an $\alpha$ probability of draws from $I$ yielding “bad” similar inputs such that, when $f$ runs on these inputs, there is a greater than $\delta$ chance that $f$ will induce more privacy cost than $\epsilon$.

**Theorem 12.** Given a fixed distribution $I$ over similar inputs, a positive integer $k$, an $(\epsilon + k\omega, \delta, \alpha)$-ill-behaved program $f$, and a positive value $\theta$, if

1. $f$ makes at most $k$ calls to the Laplace sampling instructions in one execution,
2. $f$ has at most $n$ output buckets (Section 5), and
3. DPCheck failed to reject $f$, because DPCheck discovered shift$_i$ values (used in eq. (4)) that are valid for the sampled execution traces,

then, as long as we had run at least $d$ tests with independently sampled inputs and with at least $m$ sampled traces in each test, the probability of such a failure invalidating the claim of $(\epsilon, 0)$-differential privacy for $f$ is at most $e^{-d(\theta + \alpha)}$, as long as $m \geq \frac{1}{\delta}(\theta + n k \ln 2 + n k \ln \frac{C_2 - C_1}{\omega})$ where $C_1 = \min_i$ shift$_i$ and $C_2 = \max_i$ shift$_i$.

The proof can be found in the extended version.

In practice, the shift$_i$ values are bounded by machine limits. Even if we take $C_1$ as the smallest double-precision floating point number, $C_2$ as the largest, and $\omega$ as the smallest gap between two double-precision floats, the factor $\ln \frac{|C_2 - C_1|}{\omega} = \ln \frac{1.798\times10^{308}}{2^{-52}}$ is smaller than 747, a small requirement on the number of sampled traces even in this extreme case.

Note that there is a non-zero gap of $k\omega$ between the tested privacy level $\epsilon$ for $f$ and the level $(\epsilon + k\omega, \delta, \alpha)$ to which $f$ is ill behaved. This means DPCheck is only guaranteed to catch bugs with high probability if $f$’s behavior differs enough from the claimed levels of differential privacy. However, since $\omega$ is the granularity of the discretized domain, the value $k\omega$ is typically very small. (For example, the gap between two double-precision floating point numbers in the interval $[0, 1]$ is $2^{-52}$.) Conversely, if $f$’s behavior is not very far from its claimed level of differential privacy, then we cannot give any guarantees about the probability of falsely accepting $f$.

Although we presented the testing strategy through the example algorithm $rnm$, the testing framework requires no special input particular to $rnm$. In fact, we can use the testing strategy described here to check the differential privacy property of many other algorithms, as long as these algorithms’ differential privacy proofs follow the general template listed in steps 1 to 4 and their program control flow conditions are necessary and sufficient for differential privacy. In the extended version, we describe a variety of algorithms tested using this strategy; we also present the ReportNoisyMaxWithGap algorithm, whose optimal privacy cost cannot be established using this strategy, though it can still be validated as differentially private with a non-optimal $\epsilon$. In Section 7, we discuss the details of a practical workflow that applies DPCheck to develop, test, and integrate core differential privacy mechanisms with an existing software system designed for the 2020 US Census.

**7 EVALUATION**

We seek to answer the following questions:

1. How expressive is DPCheck’s testing strategy?
2. Can DPCheck assist implementations of real-world systems that use differential privacy?
To answer these questions, we first used DPCheck to distinguish private and non-private variants of 10 differential privacy benchmark algorithms from the literature. Second, we used DPCheck in a practical workflow to re-implement and test the core differential privacy mechanism from the Disclosure Avoidance System (DAS) for the 2020 US Census [Petti and Flaxman 2019]. To save space, we concentrate on the latter experiment and give just a brief summary of the former; further details can be found in the extended version.

### 7.1 Benchmark Algorithms (Summary)

We used DPCheck to implement a suite of benchmark algorithms from the literature. For each algorithm, we built both a correct implementation and several non-differentially-private variants. We expected DPCheck to accept the correct implementation according to Lemma 11 and to detect and reject all non-differentially-private variants. The results are shown in Figure 1. We place a ✓ in the “Correct” row if DPCheck accepts the correct implementation under the algorithm’s optimal privacy cost, and we put a ✓ in the “Buggy” row if DPCheck rejects all non-differentially private variants of this algorithm. For the ReportNoisyMaxWithGap (rnmGap) algorithm, we write ✓∗ to indicate that DPCheck does not accept its correct implementation with the optimal privacy cost but does accept with twice the optimal privacy cost. Details can be found in the extended version.

We also compare the coverage over these ten benchmark algorithms between DPCheck and related frameworks in Figure 1. For each related framework, ✓ indicates that the framework in the corresponding row has successfully analyzed the algorithm in the corresponding column. A ✓? represents that the framework in the corresponding row cannot be used to test or verify the algorithm in the corresponding column. A gray ✓? indicates that the authors of this framework have not presented an evaluation of the algorithm in the corresponding column either in its publication or its released software artifact, but that we believe the framework has enough expressive power to handle this algorithm. Similarly, a gray ✓? indicates our belief that the algorithm in the corresponding column is beyond the capabilities of the framework.

The table shows that DPCheck correctly accepts private implementations and rejects faulty variants of all previous studied benchmark algorithms. Additionally, it is the first framework able to distinguish between correct and faulty variants of PrivTree.

PrivTree is challenging for automatic verification of differential privacy for at least two reasons. The first is that PrivTree terminates probabilistically, i.e., the probability of PrivTree not terminating after n iterations of its main loop diminishes as n increases. The second reason is that the privacy analysis used in the PrivTree’s privacy proof involves intermediate privacy costs that depend on input values [Zhang and Kifer 2017a].

---

**Fig. 1.** DPCheck test results and coverage comparison on benchmark algorithms with other frameworks.
The first characteristic poses issues for static analyses (including DPCheck’s symbolic interpreter), as we cannot statically know how many iterations PrivTree will run. Fortunately, as discussed in Section 5, our symbolic interpreter only needs to produce trees that match those observed in the instrumented execution. The trees produced by PrivTree contain strictly more nodes as loop iteration counts increase. Thus, our symbolic interpreter can stop searching for matching trees once it realizes that all future iterations will produce trees that cannot match those observed in the instrumented executions.

The second characteristic is a serious issue for tools aimed at automatically generating proofs of differential privacy. Since such tools need to reason over all possible input values, intermediate privacy costs that depend on inputs must be represented by expressions over these unknown inputs. For PrivTree, these intermediate privacy cost expressions involve non-linear arithmetic, an undecidable theory that can only be solved in a best-effort way by SMT solvers.

By contrast, DPCheck’s testing framework chooses a pair of concrete input values and evaluates PrivTree over these inputs. This allows the testing framework to represent intermediate privacy costs with much simpler symbolic expressions.

The two most challenging algorithms for existing validation and verification frameworks are PrivTree and ReportNoisyMaxWithGap. StatDP and DP-Finder cannot process the complex output type of PrivTree—tree data structures, since both of these tools rely on heuristics that are designed to detect DP violations on numerical outputs. LightDP and ShadowDP, the two type-system based tools, fail at PrivTree due to the unbounded probabilistically terminating main loop. We believe PrivTree is out-of-scope for the proof synthesis framework for the same reason—the proof synthesis framework would not be able to analyze an unbounded probabilistically terminating main loop.

We believe StatDP and DP-Finder’s implementations could be improved with additional heuristics to handle the output datatype of ReportNoisyMaxWithGap, while LightDP and ShadowDP require changes to their type systems. The proof synthesis artifact is unavailable, but since it is based on apRHL, and apRHL is expressive enough to prove differential privacy for ReportNoisyMaxWithGap, we believe the proof synthesis framework can handle ReportNoisyMaxWithGap.

### 7.2 Disclosure Avoidance System

Every ten years, the US Census Bureau conducts a national survey to count the total population in the United States. This survey, referred to as the Decennial Census, provides critical information for the Federal Government to adjust allocation of funds, as well as representation in the US House of Representatives, where each state gets a number of delegates proportional to its population. For the 2020 Census, the US Census Bureau developed an open-source Disclosure Avoidance System (DAS) to aggregate raw survey data into population counts. DAS applies differential privacy to protect the privacy of survey participants.

DAS aims to produce differentially private population counts for each geographical unit within each of the six geographical levels in the US: the whole nation, individual states, counties, “census tracts,” “block groups,” and single city blocks [Petti and Flaxman 2019]. This process would be straightforward if the only requirement were differential privacy: just count the population in each geographical unit and add appropriately sampled noise to each count. However, a census report produced through this idealized process would contain inconsistent counts due to the added noise: for example, the population count in a state might well be different from the sum of the population counts from all counties within the state, and there might even be negative counts for some geographical units where the precise count before adding noise was small. The Census Bureau has a list of data requirements that rules out these inconsistencies, and the final report produced by DAS must satisfy these requirements [Petti and Flaxman 2019].
To address these issues, DAS applies a so-called “TopDown Algorithm.” The TopDown Algorithm consists of 2 phases. The first phase calculates precise (and secret) counts for all geographical units, then adds appropriately sampled noise to produce noisy public counts.

The second phase iterates over the geographical hierarchy, ordered from coarsest (nation) to finest (block). Each step takes the noisy counts from two adjacent levels and perturbs them using constrained optimization. The constrained optimization process perturbs noisy counts to rule out inconsistencies, while the optimization objective keeps the overall perturbation of noisy counts as small as possible.

Since the outputs from the first phase already satisfy differential privacy, the outputs from the second phase do too (because differential privacy is robust to postprocessing [Dwork et al. 2006]). The second phase does not introduce any additional randomly sampled noise.

DAS is an interesting target for differential privacy testing due to the social importance of DAS’s privacy properties. Furthermore, by applying DPCheck on a piece of large, real-world software artifact like DAS, we gain insight on how DPCheck can assist in developing real-world software systems that interact with sensitive data through differential privacy in the future.

For this evaluation, we manually re-implemented, in DPCheck, the core privacy mechanism that calculates the scale of noise distributions and releases the noisy counts for each geographical unit. We used the DPCheck testing framework to check that this mechanism is, indeed, differentially private, and verified DPCheck can reject faulty variants of this mechanism. The faulty variants were edited from the correct implementation to simulate common programming mistakes—sampling noise with wrong parameters, and iterating over the input list with off-by-one errors. Finally, we mechanically extracted the DPCheck code to Python3 code (by pretty-printing, essentially).

Figure 2 shows the DPCheck code that re-implements the core privacy mechanism. The function geometricFixedSens takes a precise count \( \text{trueAnswer} \), a parameter \( \text{sens} \) that bounds the difference of \( \text{trueAnswer} \) between similar inputs, and the amount of privacy budget allocated for adding noise to this value \( \text{eps} \). Here, \( \text{trueAnswer} \) corresponds to the accurate and secret count of population in a geographical unit. The value of \( \text{sens} \) measures how much the precise count of this geographical unit can change between two similar inputs of the Census data; it is determined manually by Census scientists. The value of \( \text{eps} \) is also determined by Census scientists, to provide a suitable level of privacy protection. From \( \text{sens} \) and \( \text{eps} \), we can calculate the appropriate \( \alpha \) parameter of the two-sided geometric distribution and add sampled noise to \( \text{trueAnswer} \) to produce a differentially private noisy answer.

The function loopGeometricFixedSens takes a list of input values in the form

\[
(\text{trueAnswer}_1, (\text{sens}_1, \text{eps}_1)), \ldots, (\text{trueAnswer}_n, (\text{sens}_n, \text{eps}_n))
\]

and creates an output list of the same size that contains the noisy answers for each \( \text{trueAnswer}_i \) in the list. The privacy guarantee of this procedure is that it is \( (\Sigma_i \text{eps}_i, 0) \)-differentially private.

We try to experimentally detect violations of differential privacy from loopGeometricFixedSens by randomly generating similar list inputs, running DPCheck’s testing framework with the generated lists, and checking that each test reports no violations. We repeat this entire procedure in a non-terminating loop running on a cloud virtual machine that stores all test logs and raises alarms for any test failure. The size of the input lists increase with each passing test, up to 100.

We observed that the tests occasionally fail due to a tiny over-use of the \( \epsilon \)-privacy budget—in the range \( 10^{-12} \) to \( 10^{-15} \). We believe this was caused by rounding errors from the floating point operations that calculate noise distribution parameters in geometricFixedSens. We also observed that if we relax the privacy parameter to \( \Sigma_i \text{eps}_i + 10^{-12} \), then all of our test cases passed. This
suggests that the total privacy cost incurred by running \texttt{loopGeometricFixedSens} is a small value plus the sum of intended \(\epsilon_i\) values, due to rounding in floating point operations.

Indeed, we expect the original version of DAS to also have this property, since its Python\textsuperscript{3} implementation also uses floating-point arithmetic. We confirmed this conjecture by intercepting the raw inputs to the core privacy mechanism in the original version of DAS, converting these inputs into 128-bit floating point numeric representations, which preserves much more precision than the 64-bit floating point numbers used in DAS, and calculating the total privacy cost in 128-bit floats. We then compared this higher-precision total privacy cost with the total privacy cost reported by the original version of DAS. This comparison reveals that the total privacy cost is around \(1.03 \times 10^{-11}\) more than the reported total privacy cost. Of course, because the extra privacy cost is extremely small, it does not significantly degrade the privacy protections provided by DAS.

Finally, to confirm that our re-implemented privacy mechanism behaves the same as the original version, we extract \texttt{loopGeometricFixedSens} to Python\textsuperscript{3} and replace the original core mechanism with the extracted code (shown in the extended version). We then apply the following test setup to compare the behavior between two versions of DAS. For each trial:

1. Run both versions of DAS 500 times.
2. Assume that both groups of outputs come from the same distribution (since we assume both versions of DAS exhibit identical behavior); run statistical test to check if there is evidence to reject this assumption.
3. Record the \(p\)-value from the hypothesis test.

If our null hypothesis—that both versions of DAS behave identically—is true, then we should observe that the recorded \(p\)-values follow a uniform distribution on the interval \([0, 1]\) [Murdoch

Fig. 2. DAS Core Mechanism
et al. 2008]. Accordingly, we perform one final hypothesis test on the recorded \( p \)-values to search for evidence that suggests otherwise.

To run DAS, we need census data as inputs. The US Census Bureau tested DAS’s functionality using 1940 Census data [Ruggles et al. 2020]. With the 1940 Census data, each DAS run takes roughly 6 hours on our test machine. Since we need to perform 500 runs on each version of DAS per trial, and perform many trials to record enough \( p \)-values, we cannot afford to run DAS on the full 1940 Census dataset.

Instead, we subsample around 1 percent of the 1940 Census dataset and perform our trials over this smaller dataset. On our subset of the 1940 Census data, each run takes around 10 minutes to finish and produces a vector of 287509 counts for the geographical units contained in the smaller dataset. We also parallelize the trials with 12 machines to speed up the entire test process.

Since our null hypothesis is that the two versions of DAS produces the same output distribution, we need a statistical test that can invalidate our null hypothesis based on the observations of the output vectors. We use multivariate permutation testing [Chung and Romano 2016] for this task. The multivariate permutation test takes two groups of samples as inputs. In our case, the two groups are each 500 vectors, one produced by our modified version of DAS and the other produced by the original version of DAS. The test randomly swaps vectors between these two groups and compares a test statistic derived from the difference of sample mean vectors from them before and after swapping. Intuitively, if both groups of samples truly come from the same distribution, then the test statistic should not change much due to swapping.

Each run of the permutation test produces a \( p \)-value. Tests that consistently produce very small \( p \)-values are evidence invalidating our null hypothesis that two versions of DAS have identical behavior. Here, we are checking for the lack of such evidence: when our assumption is indeed true, the \( p \)-values are samples drawn uniformly at random in the interval \([0, 1]\). We perform a final Kolmogorov-Smirnov test [Massey 1951] to check if there is evidence for rejecting the hypothesis that \( p \)-values are drawn from a uniform distribution. This final test produces a \( p \)-value of 0.68, signaling a lack of evidence to reject the hypothesis that the recorded \( p \)-values are sampled from a uniform distribution. We can also plot a histogram of the observed \( p \)-values (Section 7.2), which visually shows the collected \( p \)-values. These results produce no evidence for rejecting our null hypothesis that the two versions of DAS indeed behave identically.

To summarize our evaluation of DPCheck on the DAS workflow: we successfully re-implemented the privacy mechanism used by DAS, tested its privacy properties using DPCheck’s testing framework, extracted the DPCheck implementation into Python3 code, re-integrated this extracted privacy mechanism with the rest of DAS, and confirmed statistically that this modified version of DAS behaves the same as the original. This case study demonstrates that we can develop and test core differential privacy mechanisms in DPCheck and then integrate these core procedures with large software systems through mechanized code extraction.

8 OPTIMIZATIONS

Speeding up symbolic execution. We described a program transformation, streamline, that turns DPCheck programs into straight-line programs in Section 5.4. This transformation produces exponentially many straight-line programs that each need to be analyzed, placing a bottleneck on the size of inputs we can test. To mitigate this blowup in some cases, we implemented an optimized
symbolic interpreter that does not require streamline and instead applies a type-driven state-merging algorithm developed by Torlak and Bodik [2014] This massively speeds up DPCheck’s symbolic execution and allows us to scale the generation of symbolic formulas to much larger input sizes. However, these formulas are more complex than those produced by the simpler method, and solving them is likely NP-complete. After comparing the end-to-end testing time with and without this optimization, we observed that the only algorithms that saw a speedup are the ones whose intermediate states merge well (such as ReportNoisyMax), while algorithms such as SparseVector, whose intermediate states do not merge well, performed worse than with the original approach. DPCheck’s testing framework supports both versions, so that users may take advantage of the state-merging optimizations when appropriate.

**Bucketing Double results.** As described in Section 5, the testing process involves a “bucketing” step that groups sampled traces with the same outputs. This step is easy for algorithms that yield a small number of different outputs, as we only need to perform equality tests to group the sampled traces. However, DPCheck is not limited to such algorithms. For example, SmartSum and SparseVectorGap both yield Doubles, which are computed using values sampled from Laplace distributions. It is highly unlikely that any two runs of SmartSum or SparseVectorGap will produce the same Laplace samples, even if they follow the same control flow path. So we cannot simply use equality tests to bucket the outputs.

One solution is to restrict the output types of algorithms so that they only contain a small number of possible values, but this severely limits the kinds of algorithms that can be tested with DPCheck.

Instead, DPCheck chooses a heuristic that trades off some completeness for allowing programmers to test algorithms that may return sampled Double values. At test time, DPCheck’s instrumented interpreter attaches a distribution provenance to each sampled value and to results of arithmetic expressions that involve sample values.

For example, if \( x_1 \) and \( x_2 \) are two independent samples from the Laplace distribution with center 0 and width 1, then \( x_1 \) and \( x_2 \) have distribution provenance \( \text{lap}^{1}_{0,1} \) and \( \text{lap}^{2}_{0,1} \), and an expression such as \( x_1 \cdot x_2 \) has distribution provenance \( \text{lap}^{1}_{0,1} \cdot \text{lap}^{2}_{0,1} \). The superscript allows us to distinguish the distribution provenance of \( x_1 \cdot x_2 \) from \( x_1 \cdot x_1 \) and thus recognize that these are two different distributions.

DPCheck’s testing framework then buckets output sample values based on the equality between distribution provenance structures when the output values are not equal due to independent sampling. Since most algorithms that do return sampled values only sample from a handful of possible output distributions, this heuristic significantly cuts down the number of output buckets for such algorithms.

This heuristic does not sacrifice soundness with respect to the PW-Eq proof rule. PW-Eq allows us to construct one pointwise proof for each pair of equal output values. Here, DPCheck’s heuristic still adheres to this quota; indeed, it goes a step further by posing an even more stringent quota that only allows one pointwise proof for each pair of equal output distributions. In our evaluation, this heuristic allows us to test several benchmark algorithms that return sampled Double values. Attaching distribution provenance values to sample values introduces some interpretation overhead, but this overhead is not a bottleneck in DPCheck’s testing performance in our evaluation.

9 LIMITATIONS

**Gap from Theoretical Guarantees.** Due to scaling issues, our current testing framework does not allow us to run tests large enough to give meaningful \((\epsilon, \delta, \alpha)\)-random differential privacy guarantees through Theorem 12. For example, to achieve guarantees with \( \delta = 10^{-5} \), by Theorem 12 we know we need at least \( 10^5 \) samples. Our current evaluation uses between 500 to 5000 sampled
traces per test iteration—i.e. the theoretical lower limit on sampled traces is at least 20 times larger than our current test parameter. Furthermore, the size of the formula to be solved by Z3 grows linearly with the number of sampled traces we use. Since the time it takes to solve these formulas grows exponentially in terms of formula size, testing with $10^5$ sampled traces may conservatively slow down testing time by around $k^{20}$ for some base exponent $k$. Given these scaling issues, DPCheck’s testing framework is more useful for catching differential privacy bugs than for validation.

**Numerical Implementation Issues.** We study DPCheck’s testing guarantees by assuming a countably infinite discretized domain, where consecutive points in the discretized domain are exactly $\omega$ apart. But DPCheck’s implementation uses double-precision floating point numbers. This mismatch is known to lead to privacy leaks [Ding et al. 2019; Mironov 2012]. It can be remedied by using fixed-precision numbers.

**Improving Symbolic Execution on Probabilistically Terminating Programs.** PrivTree is the only probabilistically terminating program we have tested in the evaluation. To avoid infinitely unrolling the main loop of PrivTree during its symbolic execution, we explicitly abort the computation when the outputs from symbolic execution cannot possibly be matched with those observed in instrumented execution. This is a rather ad-hoc treatment that introduces an unnecessary abort instruction in DPCheck. However, it is possible to generalize the underlying principle behind our current treatment of probabilistically terminating programs. First, the programmer needs to identify a metric over the program’s output, and must ensure that this metric is monotonically increasing as more iterations of the program are executed. Next, we can find the maximum value of this metric in the outputs among the instrumented executions, and cut off a potentially infinite unrolling in the symbolic execution when this loop metric exceeds the maximum value observed from instrumented executions.

**10 RELATED WORK**

The diverse body of related research on programming-language approaches to differential privacy can be roughly grouped into these categories.

**Axiomatic Systems.** Many languages have been designed explicitly for implementing differentially private programs. Notable examples are: (1) Fuzz [Reed and Pierce 2010], a functional programming language with a linear type system for tracking functional sensitivity and $\epsilon$-differential privacy, (2) DFuzz [Gaboardi et al. 2013], a dependently typed Fuzz that allows index-refinement types for more precise tracking of $\epsilon$-differential privacy, (3) AdaptiveFuzz [Winograd-Cort et al. 2017], a multi-stage functional programming language that supports Adaptive Composition [Rogers et al. 2016] of differential privacy, (4) and Duet [Near et al. 2019], a functional programming language that extends Fuzz to approximate differential privacy.

These languages are designed with specialized type systems that internalize differential privacy proofs of useful mechanisms. DPCheck is also a language for programming differential privacy, but it is our choice to not use a specialized type system for differential privacy in DPCheck. The datatypes in DPCheck are only simple types that help programmers avoid common mistakes such as multiplying a number by a list. By only using a standard type system, we keep DPCheck’s testing design applicable to conventional programming languages and idioms.

**Mechanized Proof Systems.** There has been a line of work on developing type and proof systems for the purpose of building machine-checkable proofs for differential privacy. Examples include: (1) LightDP [Zhang and Kifer 2017b], a dependently typed language that uses annotations to automate differential privacy proofs, (2) ShadowDP [Wang et al. 2019], an improvement over
LightDP that handles more sophisticated algorithms, with less annotation, and in less time, (3) apRHL [Barthe et al. 2016b], a program logic built on probability distribution coupling theory for manual proofs of differential privacy, (4) work by Albarghouthi and Hsu [2017] on a system that borrows from apRHL to automatically synthesize differential privacy proofs, and (5) work by Barthe et al. [2019] on a system that automatically proves and disproves differential privacy for a language restricted to finite domains.

The goal of these systems is to mechanically certify privacy of small programs with complex proofs. They all rely on general typing (or proof) rules that can capture the proofs of mechanisms such as ReportNoisyMax and SparseVector. LightDP, ShadowDP, and Albarghouthi and Hsu [2017]’s work all perform static analysis with the help of a solver; for privacy mechanisms with intermediate steps that depend on input values (e.g., PrivTree), the underlying solver would likely yield inconclusive results due to arithmetic complexities [Zhang and Kifer 2017b]. Furthermore, these systems often carry the burden of proving termination of the program under analysis. When faced with probabilistically terminating programs (e.g. PrivTree), they fail to produce useful analysis. DPCheck achieves greater expressiveness compared to these systems by considering the privacy properties of particular runs of a program, rather than trying to prove the program’s privacy property. This testing-based approach avoids the arithmetic challenge described by [Zhang and Kifer 2017b], since much of the arithmetic complexity is evaluated away early, and allows DPCheck to gracefully test probabilistically-terminating programs by limiting its symbolic exploration through instrumentation traces.

Barthe et al. [2019]’s work is unique in that the authors restrict the problem of automatic verification to a finite domain. In their model language, all data types contain a finite number of elements. This restriction allows the authors to craft complete decision procedures that prove or disprove differential privacy. DPCheck does not restrict its datatypes to finite domains, and its testing framework is not complete. Supporting programs with unusual differential privacy proof structures remains an important direction of our future research.

Since DPCheck tests a program for differential privacy, and cannot prove differential privacy. DPCheck by itself may not be sufficient for critical applications. When absolute guarantee of differential privacy is required, a more manual verification with a mechanized proof system is still necessary.

Statistical Testing. studies how to invalidate hypotheses about probability distributions; its techniques have also been applied to detect violations of differential privacy. Representative work includes: (1) StatDP [Ding et al. 2018], a framework for statistical testing of differential privacy, (2) DP-Finder [Bichsel et al. 2018], a framework that detects violations of differential privacy through code transformation, a careful sampling technique and objective optimization, and (3) work by Wilson et al. [2019], a SQL toolkit for differential privacy.

DPCheck is very similar to StatDP and DP-Finder in their goal of automatic testing of differential privacy, but they are very different in how they achieve this goal.

StatDP repeatedly runs the program under test, constructs two histograms approximating the two output distributions on similar inputs, and compares these two histograms using statistical tests to detect violations of differential privacy.

DP-Finder applies a novel sampling technique to construct a formula that approximates the privacy loss random variable, and then infers a lower bound of $\epsilon$ through objective optimization on this approximation formula.

DPCheck also repeatedly runs a program under test on one of the two similar inputs, using an instrumented interpreter that collects traces. Both StatDP and DP-Finder place restrictions on the shape of outputs from programs under tests, because both frameworks apply heuristics to detect
output events that likely indicates a violation of differential privacy. DPCheck adapts a general proof technique into a testing strategy, so that we only require equality tests on outputs of the program under test.

Wilson et al. [2019] developed an extension to the PostgreSQL database that checks differential privacy properties of SQL queries. They also applied histogram-based statistical testing to validate the correctness of their implementation of the extension.

11 FUTURE WORK

Addressing the scalability of DPCheck’s testing framework remains the most important avenue for improvement. Section 8 introduced a type-driven optimization to remove one of two significant scaling bottlenecks; this optimization speeds up symbolic execution, but checking satisfiability for the resulting formulas remains a serious bottleneck. In practice, this bottleneck prohibits DPCheck’s testing framework to run tests large enough for validation at a high confidence level.

To reduce this gap between theory and implementation, we plan on improving both sides. On the theory side, Theorem 12 only gives a very crude lower bound on the number of sampled traces required for a given confidence level; we believe it can be improved by more careful analysis. On the implementation side, (1) we can develop domain specific solver heuristics for the kinds of formulas that DPCheck generates, and (2) we can develop specification-based testing for DPCheck, approaching validation through programmer annotations of shift values rather than using Z3 to synthesize them.

Other avenues for improvement also remain. In particular, we hope to (1) harden the current implementation using fixed-precision instead of floating point numbers, (2) improve the current ad-hoc treatment of probabilistically terminating programs as discussed in Section 9, and (3) incorporate more sophisticated relations on samples to increase expressiveness.

ACKNOWLEDGMENTS

We are grateful to Danfeng Zhang, Daniel Winograd-Cort, Justin Hsu, and the Penn PLClub for discussion and comments, and we thank the anonymous reviewers for their detailed feedback. This work was supported in part by the National Science Foundation under grants CNS-1065060 and CNS-1513694.

REFERENCES


