AUTOMATED METHODS FOR CHECKING DIFFERENTIAL PRIVACY

BY

VISHAL JAGANNATH RAVI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Mahesh Viswanathan

ABSTRACT

Differential privacy is a *de facto* standard for statistical computations over databases that contain private data. The strength of differential privacy lies in a rigorous mathematical definition which guarantees individual privacy and yet allows for accurate statistical results. Thanks to its mathematical definition, differential privacy is also a natural target for formal analysis. A broad line of work uses logical methods for proving privacy. However, these methods are not complete, and only partially automated. A recent and complementary line of work uses statistical methods for finding privacy violations. However, the methods only provide statistical guarantees (but no proofs).

We propose the first decision procedure for checking differential privacy of a non-trivial class of probabilistic computations. Our procedure takes as input a program P parametrized by a privacy budget ϵ and either proves differential privacy for all possible values of ϵ, or generates a counterexample. In addition, our procedure applies both to ϵ-differential privacy and (ϵ, δ)-differential privacy. Technically, the decision procedure is based on a novel and judicious encoding of the semantics class of programs in our class into a decidable fragment of the first-order theory of the reals with exponentiation. We implement our procedure and use it for (dis)proving privacy bounds for many well known examples, including randomized response, histogram, report noisy max and sparse vector.

To my parents, for their love and support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Differential privacy [1] is a gold standard for privacy of statistical computations. Differential privacy ensures that running the algorithm on any two "adjacent" databases yields two "approximately" equal distributions, where two databases are adjacent if they differ in a single element, and two distributions are approximately equivalent if their distance is small w.r.t. some specific metric. Thus, differential privacy delivers a very strong form of individual privacy. Yet, and somewhat surprisingly, it is possible to develop differentially private algorithms for many tasks. Moreover, the algorithms are useful, in the sense that their results have reasonable accuracy. However, designing differentially private algorithms is difficult and the privacy analysis can be error-prone, as witnessed by the example of sparse vector. This difficulty has motivated the development of formal approaches for analyzing differentially private algorithms; see [2] for a survey and the related work section of this report.

Even though significant advances have been made in identifying proof principles to establish differential privacy [3, 4, 5, 6, 7, 8, 9, 10] and techniques have been proposed to find differential privacy violations [11, 12], basic questions — Is checking differential privacy decidable? What are the limits of automated checking of differential privacy? What is the asymptotic complexity of checking? — have (shockingly) remained unanswered. The sole metric for evaluating proposed checking techniques has been experimental evaluation on examples rather than a precise mathematical characterization of its limits. However, if past experience in automated formal verification of other application areas is any guide, then answering these foundational questions is essential for principled algorithmic development that is not only critical for theoretical advances but also for building practical tools that scale to large examples.

This report is a first attempt at addressing this serious lacuna in the current state of understanding in formal verification of differential privacy. Our first result establishes that checking differential privacy is indeed, computationally, a very difficult problem. We show that the problem of checking differential privacy is undecidable. Our proof of this result shows that the problem remains undecidable even if one considers programs with a *single* Boolean input, and a *single* Boolean output.

The main thrust of this report is, therefore, to identify a rich class of programs, that encompasses many known examples, for which checking differential privacy is decidable for all possible instances of the privacy parameter ϵ (throughout the report, we assume that the error parameter δ is defined as a function of ϵ). Our undecidability proof highlights the

challenges in this enterprise — since it applies to really simple programs, can we even hope to find a practically relevant decidable fragment? We focus our attention on programs whose input and output spaces are finite. Note that such programs need not be finite state, as they could use auxiliary, variables, ranging over infinite domains, to carry out the computation. We introduce a class of programs, called DiPWhile, which are probabilistic while programs, for which the problem of checking differential privacy is decidable. We succeed in carefully balancing the twin (orthogonal) demands of decidability and expressivity, by judiciously delineating the use of real-valued and integer-valued variables. Our decidability proof for programs in DiPWhile has the following salient features. The first step is an observation that the semantics of DiPWhile-programs can be defined using parametrized, *finite-state* Markov chains [1]. The fact that the semantics can be defined using only finitely-many states is a surprising observation because our programs have integer and real-valued variables. Our key insight here is that a precise semantics for DiPWhile-programs can be given *without* tracking the explicit values of the real and integer-valued variables. Second, we observe that the transition probabilities of the Markov chain semantics are pseudo-rational functions of the privacy budget. These two observation together, allow us to reduce the problem of checking differential privacy of DiPWhile-programs to the decidable fragment of the first order theory of reals with exponentials, identified by McCallum and Weispfenning [13].

Our decision procedure has two complementary uses. The first use of the procedure is a stand alone tool for checking ϵ- or $(\epsilon, \delta(\epsilon))$-differential privacy of mechanisms specified by DiPWhile-programs, for all values of ϵ. We have implemented our decision procedure in a tool that we call DiPC (**Di**fferential **P**rivacy **C**hecker). Given DiPWhile-program, our tool constructs a sentence in McCallum-Weispfenning fragment of the theory of reals with exponentials. It then calls Mathematica® to check if the constructed sentence is true over the reals. Since our decision procedure is the *first* that can both prove differential privacy and detect its violation, we tried the tool on examples that known to be differentially private and those that are known to be not differentially private, including variants of Sparse Vector, Report Noisy Max, and Histograms. DiPC successfully checked differential privacy for the former class of examples and produced counterexamples for the later class. Our counterexamples are exact (rather than probabilistic) and are more compact than those delivered by prior tools.

A complementary use of the decision procedure is for validating counter examples for algorithms with infinite input or output sets. Our approach can be used to check ϵ-differential privacy of any mechanism for a given pair of adjacent input values and a given output value,

[1] A parametrized Markov chain is a Markov chain whose transition probabilities are a function of the privacy budget.

for all values of $\epsilon > 0$, or for a given value of $\epsilon > 0$. It can also be used to find violations of programs with unbounded outputs. For such programs, it is possible to discretize the output domain into a finite domain, and to use the decision procedure to find privacy violations for the discretized algorithm (by post-processing, privacy violations for the discretized algorithms are also privacy violations for the original algorithm). Such usages complement the technique presented in [11] that proposes a method for generating counter examples. The approach of [11] is statistical in nature, and the examples generated by it are highly probable to be counter examples (with statistical guarantees), but may not be definite counter examples. Our approach can be used to check if the counter examples, generated by their tool, are real counter examples, for a given value of ϵ.

1.1 CONTRIBUTIONS

We summarize our key contributions.

- We prove the undecidability of the problem of checking differential privacy of very simple programs, including those that have a single Boolean input and output.
- We identify an expressive class of programs and give the *first sound and complete* method to verify differential privacy for a class of programs. That is, we present the first single, *fully automatic* method that can prove both prove differential privacy *and* detect its violation by generating *counterexamples*; hitherto all previous approaches can either only prove differential privacy or only prove its violation, but not do both.
- We implement the decision procedure and evaluate our approach on private and non-private examples of the literature.

1.2 TOOL AND EXPERIMENTS

The tool and experiments are available at the anonymous url [14].

1.3 RELATED WORK

The main thread of work has focused on formal systems for proving that an algorithm is differentially private. Such systems are helpful because they rule out the possibility of mistakes in privacy analyses. Reed and Pierce [3] propose the first programming language technique for proving differential privacy, in the form of a linear type system. Gaboardi et

al [4] later enrich their approach with linear dependent types, in order to support recursion and a broader set of differentially private constructions. Azevedo de Amorim et al [9] propose another extension to accommodate (ϵ, δ)-differential privacy. However, it is not possible to verify some of the most advanced examples, such as sparse vector or vertex cover, using these type systems. Moreover, type-checking and type-inference for linear (dependent) types is challenging. Barthe et al [5, 6, 7] develop several program logics for reasoning about differential privacy. These logics construct approximate probabilistic couplings between two program executions on adjacent inputs. These couplings are parametrized by a binary relation on program outputs; when specialized to the equality relations, these approximate probabilistic couplings coincide with the notion of approximate equality used in differential privacy. These logics have been used successfully to analyze many classic examples from the literature, including the sparse vector technique. However, these logics are limited: they cannot disprove privacy; extensions may be required for specific examples; building proofs is challenging. The last issue has been addressed by Zhang and Kifer [8] and by Albarghouthi and Hsu [10]. These works propose automated methods for proving automatically differential privacy. Zhang and Kifer introduce randomness alignments as an alternative to couplings, and build a dependent type system that tracks randomness alignments. Automation is then achieved by type inference. Albarghouthi and Hsu propose coupling strategies, which rely on a fine-grained notion of variable approximate coupling which draws inspiration both from approximate couplings and randomness alignment. They synthesize coupling strategies by considering an extension of Horn clauses with probabilistic coupling constraints, and developing algorithms to solve such constraints. However, these methods are limited to vanilla ϵ-differential privacy and do not accommodate bounds that are obtained by advanced composition (since $\delta \neq 0$). Recently, Liu, Wang, and Zhang [15] develop a probabilistic model checking approach for verifying differential properties. Their approach is based on modelling differential private programs as Markov chains. Their encoding is more direct than ours (i.e. it does not attempt to build a finite-state Markov chain) and they do not provide a decision procedure. Chistikov and Murawski and Purser [16] propose an elegant method based on skewed Kantorovich distance for checking differential privacy of Markov chains. However, their approach is rather theoretical and not implemented.

A dual problem is to automatically find violations of differential privacy. This would help privacy practitioners discover potential problems in their algorithms as early as possible. Two recent and concurrent works by Ding et al [11] and Bischel et al [12] develop automated methods for finding privacy violations. Ding et al propose an approach which combines purely statistical methods based on hypothesis testing and symbolic execution. Bischel et al develop an approach based on a combination of optimization methods and language-specific

techniques for computing differentiable approximations of privacy estimations. Both methods are fully automated. However, their guarantees are statistical (no proofs). Moreover, both methods can only be used for concrete numerical values of the privacy budget ϵ. As previously explained, our work is complementary to these approaches, in the sense that our decision procedure can be used to verify their proposed counter-examples (for algorithms that fall in the class of programs handled by the procedure).

To our best knowledge, no prior work is able to both prove differential privacy and detect its violations for a non-trivial class of programs.

Our work is also loosely connected to prior attempts to relate differential privacy and information flow. In particular, Barthe and Köpf [17] study information-theoretic bounds for differentially private channels and provide a decision procedure for rational bounds. Their decision procedure is based on a reduction to the theory of real closed fields (without exponential). However, there approach considers channels and is not directly applicable to a language-based setting.

CHAPTER 2: PRIMER ON DIFFERENTIAL PRIVACY

Differential privacy [1] is a rigorous definition and framework for private statistical data mining. In this model, a trusted curator with access to the database returns answers to queries made by possibly dishonest data analysts that do not have access to the database. The curator's task is to return probabilistically noised answers, so that data analysts cannot distinguish between two databases which are adjacent, i.e. only differ in the value of a single individual. There are two common definitions: two databases are adjacent if they are exactly the same except for the presence or absence of one record, or exactly the same except for the difference in one record. We abstract away from any particular definition of adjacency.

Henceforth, we denote the set of real numbers, rational numbers, natural numbers and integers by $\mathbb{R}, \mathbb{Q}, \mathbb{N}$, and $\mathbb{Z}$ respectively. The Euler constant shall be denoted by e. We assume given a set $\mathcal{U}$ of inputs, and a set $\mathcal{V}$ of outputs. A randomized function P from $\mathcal{U}$ to $\mathcal{V}$ is a function that takes an input in $\mathcal{U}$ and returns a distribution over $\mathcal{V}$. For a measurable set $S \subseteq \mathcal{V}$, the probability that the output of P on u is in the set S shall be denoted by $\mathsf{Prob}(P(u) \in S)$. In the case the output set is discrete, we use $\mathsf{Prob}(P(u) = v)$ as shorthand for $\mathsf{Prob}(P(u) = \{v\})$.

We are now ready to define differential privacy. We assume that $\mathcal{U}$ is equipped with a binary *symmetric* relation $\Phi \subseteq \mathcal{U} \times \mathcal{U}$, which we shall call the *adjacency relation*. We say that $u_1, u_2 \in \mathcal{U}$ are *adjacent* if $(u_1, u_2) \in \Phi$.

Definition 2.1. Let $\epsilon \geq 0$ and $0 \leq \delta \leq 1$. Let $\Phi \subseteq \mathcal{U} \times \mathcal{U}$ be an adjacency relation. Let P be a randomized function with inputs from $\mathcal{U}$ and outputs in $\mathcal{V}$. We say that P is (ϵ, δ)-differentially private with respect to Φ if for all measurable subsets $S \subseteq \mathcal{V}$ and $u, u' \in \mathcal{U}$ such that $(u, u') \in \Phi$,

$$\mathsf{Prob}(P(u) \in S) \leq e^{\epsilon} \mathsf{Prob}(P(u') \in S) + \delta$$

As usual, we say that P is ϵ-differentially private iff it is $(\epsilon, 0)$-differentially private. If the output domain is discrete, it is equivalent to require that for all $v \in \mathcal{V}$ and $u, u' \in \mathcal{U}$ such that $(u, u') \in \Phi$,

$$\mathsf{Prob}(P(u) = v) \leq e^{\epsilon} \mathsf{Prob}(P(u') = v)$$

Differential privacy is preserved by post-processing. Formally, if P is an (ϵ, δ)-differentially private computation from $\mathcal{U}$ to $\mathcal{V}$, and $h : \mathcal{V} \to \mathcal{W}$ is a deterministic function, then $h \circ P$ is an (ϵ, δ)-differentially private computation from $\mathcal{U}$ to $\mathcal{W}$. In the remainder, we shall exploit post-processing to connect differential privacy of randomized computations with infinite

output spaces to differential privacy of their discretizations.

Laplace Mechanism

The Laplace mechanism [1] achieves differential privacy for numerical computations by adding random noise to outputs. Given $\epsilon > 0$ and mean μ, let $\mathsf{Lap}(\epsilon, \mu)$ be the continuous distribution whose probability density function (p.d.f.) is given by

$$f_{\epsilon,\mu}(x) = \frac{\epsilon}{2}\, e^{-\epsilon|x-\mu|}.$$

$\mathsf{Lap}(\epsilon, \mu)$ is said to be the *Laplacian distribution* with mean μ and scale parameter $\frac{1}{\epsilon}$. Consider a real-valued function $q : \mathcal{U} \to \mathbb{R}$. Assume that q is k-sensitive w.r.t. an adjacency relation Φ on $\mathcal{U}$, i.e. for every pair of adjacent values u_1 and u_2, $|q(u_1) - q(u_2)| \leq k$. Then the computation that maps u to $\mathsf{Lap}(\frac{\epsilon}{k}, q(u))$ is ϵ-differentially private.

It is sometimes convenient to consider the discrete version of the Laplace distribution. Given $\epsilon > 0$ and mean μ, let $\mathsf{DLap}(\epsilon, \mu)$ be the discrete distribution on $\mathbb{Z}$, the set of integers, whose probability mass function (p.m.f.) is given by

$$f_{\epsilon,\mu}(i) = \frac{1 - e^{-\epsilon}}{1 + e^{-\epsilon}}\, e^{-\epsilon|i-\mu|}.$$

$\mathsf{DLap}(\epsilon, \mu)$ is said to be the *discrete Laplacian distribution* with mean μ and scale parameter $\frac{1}{\epsilon}$. The discrete Laplace mechanism achieves the same privacy guarantees as the continuous Laplace mechanism.

Exponential mechanism

The Exponential mechanism [18] is used for making non-numerical computations private. The mechanism takes as input a value u from some input domain and a scoring function $F : \mathcal{U} \times \mathcal{V} \to \mathbb{R}$ and outputs a discrete distribution over $\mathcal{V}$. Formally, given $\epsilon > 0$ and $u \in \mathcal{U}$, the discrete distribution $\mathsf{Exp}(\epsilon, F, t)$ on $\mathcal{V}$ is given by the probability mass function:

$$h_{\epsilon,F,t}(v) = \frac{e^{\epsilon F(t,v)}}{\sum_{v \in \mathcal{V}} e^{\epsilon F(t,v)}}.$$

Suppose that the scoring function is k-sensitive w.r.t. some adjacency relation Φ on $\mathcal{U}$, i.e., for all for each pair of adjacent values u_1 and u_2 and $v \in \mathcal{V}$, $|F(u_1, r) - F(u_2, r)| \leq k$. Then the exponential mechanism is $(2k\epsilon, 0)$-differentially private w.r.t. Φ.

CHAPTER 3: MOTIVATING EXAMPLES

Before presenting the mathematical details of our results, let us informally present our method by showing how it would work on some illustrative examples. Consider the Sparse Vector Technique (SVT) [19, 20]. The Sparse Vector Technique was designed to answer multiple Δ-sensitive numerical queries in a differentially private fashion. The relevant information we want from queries is, which amongst them are above a threshold T. If we apply a differentially private mechanism to answer each one of them separately then the privacy budget explodes (answering k such queries in an ϵ-differentially private manner would only be $k\epsilon$-differentially private). The SVT as given in Algorithm 3.1 is designed to identify the first c queries that are above the threshold T in an ϵ-differentially private fashion.

Input: $q[1:N]$
Output: $out[1:N]$

$\mathsf{r}_T \leftarrow \mathsf{Lap}(\frac{\epsilon}{2\Delta}, T)$
$count \leftarrow 0$
for $i \leftarrow 1$ **to** N **do**
  $\mathsf{r} \leftarrow \mathsf{Lap}(\frac{\epsilon}{4c\Delta}, q[i])$
  $\mathsf{b} \leftarrow \mathsf{r} \geq \mathsf{r}_T$
  if b **then**
    $out[i] \leftarrow \top$
    $count \leftarrow count + 1$
    if $count \geq c$ **then**
      exit
    end
  else
    $out[i] \leftarrow \perp$
  end
end

Algorithm 3.1: SVT algorithm (SVT1)

In the program, the integer N represents the total number of queries and the array q of length N represents the answers to queries. The array out represents the output array, $\perp$ represents False and $\top$ represents True. We assume that initially the constant $\perp$ is stored at each position in out. In the SVT technique, the true answers account for most of the privacy cost and we can only answer c of them until we run out of the privacy budget [19, 8]. On the other hand, there is no restriction to the false answers that can be given.

The input set $\mathcal{U}$ in this context is the set of N length vectors q, where the kth element

$q[k]$ represents the answer to the kth query on the original database. The adjacency relation Φ on inputs is defined as follows: q_1 and q_2 are adjacent if and only if $|q_1[i] - q_2[i]| \leq 1$ for each $1 \leq i \leq N$.

Let us consider an instance of the SVT algorithm when $T = 0$, $N = 2$, $\Delta = 1$ and $c = 1$. Let us assume that all array elements in q come from the domain $\{0, 1\}$. In this case, we have four possible inputs $[0, 0], [0, 1], [1, 1]$, and $[1, 0]$, and three possible outputs $[\bot, \bot], [\top, \bot]$, and $[\bot, \top]$. Our approach is to compute, for each input x and output y, the probability of returning y when the input is x. Note that this probability depends on the parameter ϵ, and so what we are looking for is a *symbolic* representation of this function. For example, the probability of outputting $[\bot, \top]$ on input $[0, 1]$ is

$$r_1 = \frac{24e^{\frac{3\epsilon}{4}} - 1 + 8e^{\frac{\epsilon}{4}} + 21e^{\frac{\epsilon}{2}}}{48e^{\frac{3\epsilon}{4}}}.$$

Similarly, when the input is $[1, 1]$ and the output is $[\bot, \top]$, the probability is given by

$$r_2 = \frac{-22 + 32e^{\frac{\epsilon}{4}} - 3\epsilon}{48e^{\frac{\epsilon}{2}}}.$$

Our goal is to compute expressions like r_1 and r_2 automatically from the program, input, and output. Having computed such expressions, checking ϵ-differential privacy requires one to determine if

$$\text{for all } \epsilon > 0.\ (r_1 \leq e^{\epsilon} r_2) \text{ and for all } \epsilon > 0.\ (r_2 \leq e^{\epsilon} r_1).$$

Notice that the above conditions can be encoded as a first order sentence with exponentials, and checking if ϵ-differential privacy holds, reduces to determining if such a first order sentence is true for reals, with the standard interpretation of multiplication, addition, and exponentiation. Whether there is a decision procedure that can determine the truth of first order sentences involving exponentials over the reals, is a long standing open problem. However, certain decidable fragments of such an extended first order theory have been identified. Our main result shows that for many examples, checking differential privacy can be reduced to a decidable fragment identified by McCallum and Weispfenning [13].

Notice that if one can compute expressions for the probability producing certain outputs on a given input, we could use the above ideas to also check (ϵ, δ)-differential privacy, instead of just ϵ-differential privacy. The only change would be to account for δ in our constraints, and to consider all possible subsets of outputs, instead of just individual output values. Thus, the methods proposed here go beyond the scope of most automated approaches, which are

restricted to vanilla ϵ-differential privacy.

3.1 FINITE DISCRETIZATION OF INFINITE OUTPUT SPACES

As outlined in the introduction, our decision procedure checks differential privacy of programs whose output space is finite. In many examples, the program outputs are reals or unbounded integers (and combinations thereof). Nevertheless, we argue that our decision procedure can still be used in the verification of differential privacy. Our approach in such cases is to discretize the output space into finitely many intervals.

We illustrate this for the special case when a program P outputs the value of one real random variable, say r. Now, suppose that we modify P to output a finite discretized version of r as follows. Let $\mathsf{seq} = a_0 < a_1 < \ldots a_n$ be a sequence of rationals and let

$$\mathsf{Disc}_{\mathsf{seq}}(x) = \begin{cases} a_0 & x \leq a_0 \\ a_1 & a_0 < x \leq a_1 \\ \vdots & \\ a_{n-1} & a_{n-2} < x \leq a_{n-1} \\ a_n & \text{otherwise} \end{cases}.$$

Consider the program $P_{\mathsf{Disc,seq}}$ that instead of outputting r, outputs $\mathsf{Disc}_{\mathsf{seq}}(\mathsf{r})$. It is easy to see that if P is differentially private then so must be $P_{\mathsf{Disc,seq}}$. Therefore, if $P_{\mathsf{Disc,seq}}$ is not differentially private then we can conclude that $P_{\mathsf{Disc,seq}}$ is not differentially private. Our decision procedure is both sound and complete for a class of programs we identify. Therefore, we can use our method to find counterexamples; counterexamples for us is a pair of adjacent inputs and a value of ϵ that violates the differential privacy inequation. Thus, if our decision procedure finds a counterexample for $P_{\mathsf{Disc,seq}}$, then it also has proved that P is not differentially private. Our method can, therefore, be used as an under-approximation technique for checking differential privacy of P. In fact, it is *complete* under-approximation method in the sense that P is differentially private iff for each possible seq, $P_{\mathsf{Disc,seq}}$ is differentially private.

Let us illustrate the discretization approach to detecting privacy violations through an example. One variant of SVT algorithm is one in which the algorithm outputs noisy queries that are above the noisy threshold (and not just which queries are above the threshold). This algorithm outputs real values and is known to violate differential privacy [20]. As discussed above, while we cannot model this algorithm directly in our framework, we can model its

```
Input: q[1 : N]
Output: out[1 : N]

r_T ← Lap(ε/2Δ, T)
count ← 0
for i ← 1 to N do
    r ← Lap(ε/4cΔ, q[i])
    b ← r ≥ r_T
    if b then
        out[i] ← Disc_seq(r)
        count ← count + 1
        if count ≥ c then
            exit
        end
    else
        out[i] ← ⊥
    end
end
```

Algorithm 3.2: Discretized SVT algorithm that outputs noisy queries above noisy threshold (SVT3)

discretized version. The discretized version is given in Algorithm 3.2.

Consider the instance of this discretized algorithm with $N = 5, c = 1, \Delta = 1, T = 0$ and let *seq* consist of a single rational number 0. Consider input $i_1 = [0, 0, 0, 0, 0]$ and output $o = [\perp, \perp, \perp, \perp, 0]$. The probability of producing o on input i_1 is $p_1 = \frac{1}{1344}$. On the other hand, the probability that o is produced on input $i_2 = [1, 1, 1, 1, 0]$ is $p_2 = \frac{e^{-\frac{5\epsilon}{4}}}{1344}$. Now ϵ-differential privacy would require that

$$\mathsf{diff} = p_1 - e^{\epsilon} p_2 = \frac{1}{1344} - \frac{e^{-\frac{\epsilon}{4}}}{1344}$$

be ≤ 0. However, this is not true, for example, when $\epsilon = 27$. This counterexample was found automatically by our tool DiPC.

We conclude this chapter by pointing out that the discretization technique also allows us to complement existing statistical techniques for finding counterexamples to differential privacy. Such statistical techniques [11] assume a fixed ϵ and typically produce a candidate counterexample which is pair of adjacent inputs in_1, in_2 and an output set S (usually an interval $I_{\mathsf{out}} = (a, b)$). These techniques do not provide a method for checking whether this is a real counterexample or just a statistical anomaly. Our methods can then serve to check that this candidate counterexample is really a counterexample by taking $\mathsf{seq} = a < b$.

CHAPTER 4: CHECKING DIFFERENTIAL PRIVACY

We consider the problem of verifying the differential privacy of randomized algorithms. Typically, such algorithms are modeled as a program P_ϵ parametrized by a value ϵ. Having a paramterized program P_ϵ captures the fact that program's behavior depends on the privacy budget ϵ, with the intention of guaranteeing that P_ϵ is $(f(\epsilon), g(\epsilon))$-differentially private, where f and g are some functions of ϵ. The parameter ϵ is assumed to belong to some (potentially unbounded) interval $I \subseteq \mathbb{R}^{>0}$ with rational endpoints; usually, we take ϵ to just belong to the interval $(0, \infty)$. The program P_ϵ will be assumed to terminate with probability 1 for every value of ϵ (in the appropriate interval).

We will assume that our programs take inputs from a set $\mathcal{U}$ and produce outputs over a set $\mathcal{V}$. In this report, we will assume that both $\mathcal{U}$ and $\mathcal{V}$ are *finite* sets that can be effectively enumerated. Despite our restriction to finite input and output sets, as we will see in the next section (Section 4.1), the computational problem checking differential privacy is challenging. At the same time, the decidable subclasses we identify (Sections 4.2 and 5), are rich enough to model most known differential privacy mechanisms even though they have finite input and output sets. Extending our decidability results to subclasses of programs that have infinite input and output sets, is a non-trivial open problem at this time.

The computational problems we consider in this report are as follows. Since our programs take inputs from a finite set $\mathcal{U}$, we assume that the adjacency relation $\Phi \subseteq \mathcal{U} \times \mathcal{U}$ is given to us as an explicit list of pairs. In general, when discussing (ϵ, δ)-differential privacy of some mechanism, the additive parameter δ needs to be a function of ϵ. To define the computational problem of checking differential privacy, the function $\delta : \mathbb{R}^{>0} \to [0, 1]$ must be given as input. We, therefore, assume that this function δ has some finite representation; if δ is the constant δ_0 then we represent δ simply by the number δ_0. There are two computational problems we consider in this report.

Fixed Parameter Differential Privacy Given a problem P_ϵ over inputs $\mathcal{U}$ and outputs $\mathcal{V}$, adjacency relation $\Phi \subseteq \mathcal{U} \times \mathcal{U}$, and rational numbers $\epsilon_0, \delta_0, t \in \mathbb{Q}^{>0}$, determine if P_{ϵ_0} is $(t\epsilon_0, \delta_0)$-differentially private with respect to Φ.

Differential Privacy Given a program P_ϵ over inputs $\mathcal{U}$ and outputs $\mathcal{V}$, interval $I \subseteq \mathbb{R}^{>0}$, $\delta : \mathbb{R}^{>0} \to [0, 1]$, an adjacency relation $\Phi \subseteq \mathcal{U} \times \mathcal{U}$, and a rational number $t \in \mathbb{Q}$, determine if P_ϵ is $(t\epsilon, \delta(\epsilon))$-differentially private with respect to Φ *for every* $\epsilon \in I$.

Observe that the Fixed Parameter Differential Privacy problem can be trivially reduced to the Differential Privacy problem. Thus, an algorithm for checking Differential Privacy can

be used to solve Fixed Parameter Differential Privacy. Unfortunately, the Fixed Parameter Differential Privacy problem is extremely challenging — we will show that it is undecidable, and therefore, so is the Differential Privacy problem. We will conclude this chapter by identifying semantic conditions under which the Differential Privacy problem (and therefore the Fixed Parameter Differential Privacy problem) is decidable.

4.1 UNDECIDABILITY OF DIFFERENTIAL PRIVACY

The main result in this section is that the Fixed Parameter Differential Privacy problem is undecidable. Consider simple while programs that have variables storing Booleans and integers. Program statements are either assignments, random assignments that sample from discrete Laplacians, conditional statements, and loops. Let us denote this class of programs by Simple; in the interests of space, we skip the formal definition of such programs, relying instead on the reader's informal understanding. Inputs to such programs (i.e. set $\mathcal{U}$) are valuations to input program variables. We have the following undecidability result.

Theorem 4.1. The Fixed Parameter Differential Privacy problem and the Differential Privacy problem for programs P_ϵ in Simple is undecidable.

Proof. We shall prove this by reducing the non-halting problem for deterministic 2-counter Minsky machines (which is known to be undecidable) to the Fixed Parameter Differential Privacy problem.

Recall that a 2-counter Minsky Machine is tuple $\mathcal{M} = (Q, q_s, q_f, \Delta^1_{inc}, \Delta^2_{inc}, \Delta^1_{jzdec}, \Delta^2_{jzdec})$ where

- Q is a finite set of control states.
- $q_s \in Q$ is the initial state.
- $q_f \in Q$ is the final state.
- $\Delta^i_{inc} \subseteq Q \times Q$ is the increment of counter i for $i = 1, 2$.
- $\Delta^i_{jzdec} \subseteq Q \times Q \times Q$ is the conditional jump of counter i for $i = 1, 2$.

$\mathcal{M}$ is said to be deterministic if from each state q, there is at most one transition out of q. The semantics of $\mathcal{M}$ is defined in terms of a transition system $(Conf, (q_s, 0, 0), \rightarrow)$ where $Conf = Q \times \mathbb{N} \times \mathbb{N}$ is the set of configurations, $(q_s, 0, 0)$ is the initial configuration and $\rightarrow$ is defined as follows:

$(q,i,j) \to (q', i+1, j)$ if $(q,q') \in \Delta^1_{inc}$,
$(q,i,j) \to (q', i, j+1)$ if $(q,q') \in \Delta^2_{inc}$,
$(q,i,j) \to (q', i, j)$ if $i = 0$ and $(q,q',q'') \in \Delta^1_{jzdec}$,
$(q,i,j) \to (q'', i-1, j)$ if $i \neq 0$ and $(q,q',q'') \in \Delta^1_{jzdec}$,
$(q,i,j) \to (q', i, j)$ if $j = 0$ and $(q,q',q'') \in \Delta^2_{jzdec}$, and
$(q,i,j) \to (q'', i, j-1)$ if $j \neq 0$ and $(q,q',q'') \in \Delta^2_{jzdec}$.

We show that given a 2-counter Minsky Machine $\mathcal{M}$, there is a program $P^{\mathcal{M}}_{\epsilon} \in \mathsf{Simple}$ such that for each $\epsilon > 0$,

(a) $P^{\mathcal{M}}_{\epsilon}$ has only one Boolean input $\mathsf{b_{in}}$ and one Boolean output $\mathsf{b_{out}}$.

(b) $P^{\mathcal{M}}_{\epsilon}$ terminates with probability 1.

(c) $P^{\mathcal{M}}_{\epsilon}$ is $(\epsilon, 0)$-differentially private w.r.t adjacency relation $\Phi = \{(\mathsf{true}, \mathsf{false}), (\mathsf{false}, \mathsf{true})\}$ if and only if $\mathcal{M}$ does not halt.

A sequence of configurations $s_0, s_1, \ldots s_k$ is said to be a computation of $\mathcal{M}$ if $s_0 = (q_s, 0, 0)$ and $s_i \to s_{i+1}$ for $i = 0, 1, \ldots k-1$. A computation $s_0, s_1, \ldots s_k$ is said to be a halting computation of $\mathcal{M}$ if $s_k = (q_f, i, j)$ for some $i, j \in \mathbb{N}$.

Given a 2-counter Machine $\mathcal{M}$, $P^{\mathcal{M}}_{\epsilon}$ is constructed as follows. First we observe that without loss of generality we can assume that the set of states Q of $\mathcal{M}$ can are integers $1, 2, \ldots, m$ where m is the number of states in Q. Thus, a configuration of $\mathcal{M}$ can be encoded as three integers $\mathsf{state}, \mathsf{cntr}_1, \mathsf{cntr}_2$. The transition relations $\Delta^1_{inc}, \Delta^2_{inc}, \Delta^1_{jzdec}$ and Δ^2_{jzdec} can be encoded using 3 additional counters, $\mathsf{n\_state}, \mathsf{n\_cntr}_1, \mathsf{n\_cntr}_2$, which model the next state and the the next counter values. For example, the transition $(q, q', q'') \in \Delta^1_{jzdec}$ can be encoded using conditional statements as follows:

```
if    (cntr_1 = 0) and (state = q)
      then n_state = q';
end
if    (cntr_1 > 0) and (state = q)
      then n_state = q''; n_cntr_1 = n_cntr_1 − 1;
end
```

Let $s_1, s_2, \ldots, s_n$ be the statements encoding the transition relation. Consider the program $P^{\mathcal{M}}_{\epsilon}$ given in Figure 4.1. The program $P^{\mathcal{M}}_{\epsilon}$ initially samples k from a discrete Laplacian. If the sampled value is < 0 then it outputs false. Otherwise, it simulates $\mathcal{M}$ upto k. At the

```
Input: b_in
Output: b_out

b_out ← false
k ← DLap(ϵ, 0)
if k > 0 then
    state ← q_s
    cntr_1 ← 0
    cntr_2 ← 0
    for steps ← 1 to k do
        s_1
        ⋮
        s_n
        state ← n_state
        cntr_1 ← n_cntr_1
        cntr_2 ← n_cntr_2
    end
    if (state = q_f) and (b_in = true) then
        b_out ← true
    end
end
```

Algorithm 4.1: Program $P_\epsilon^{\mathcal{M}}$ simulating 2-counter machine $\mathcal{M}$

end of the simulation, if the halting state is reached and the input is true then it outputs true. Otherwise, it outputs false.

Clearly, $P_\epsilon^{\mathcal{M}}$ satisfies properties (a) and (b) above. That the program $P_\epsilon^{\mathcal{M}}$ has property (c) above follows from the following observations:

1. If $\mathcal{M}$ does not halt then $P_\epsilon^{\mathcal{M}}$ outputs false with probability 1.

2. If $\mathcal{M}$ halts then $P_\epsilon^{\mathcal{M}}$ outputs true with non-zero probability on input true and outputs true with zero probability on input false.

This shows that Fixed Parameter Differential Privacy is undecidable. Undecidability of Fixed Parameter Differential Privacy is obtained by taking ϵ_0 to be any constant rational number, say $\frac{1}{2}$. *q.e.d*

This theorem shows that Differential Privacy is undecidable. Undecidability of Fixed Parameter Differential Privacy is obtained by taking ϵ_0 to be any constant rational, say $\frac{1}{2}$.

4.2 A TRACTABLE SEMANTIC CLASS OF PROGRAMS

Since the problem of checking differential privacy is undecidable (Theorem 4.1) it is necessary to restrict the class of programs to get a decision procedure. The program $P_\epsilon^{\mathcal{M}}$ constructed in the proof of Theorem 4.1 is an extremely simple program. This demonstrates how challenging the task of identifying a tractable, yet useful, subset of programs is. In this section, we identify a simple *semantic* restriction under which the Fixed Parameter Differential Privacy and Differential Privacy problems are decidable. In Chapter 5, we will use the results of this section to prove that the problem of checking differential privacy is decidable for programs written in our DiPWhile language. Our goal in identifying a semantic restriction is to reduce the problem of checking differential privacy to checking the truth of a first order formula involving exponentials about the reals. Decidability of the theory of reals with exponentials is a long standing open problem, related to Schanuel's conjecture. Some fragments of this theory are known to be decidable. In particular, McCallum and Weispfenning [13] have identified a decidable fragment of the first order theory of reals with exponentials, that we will exploit. Therefore, before defining our semantic restriction, we introduce this decidable first order theory.

We will consider first order formulas over a restricted signature and vocabulary. We will denote this collection of formulas as the language $\mathcal{L}_{\mathsf{exp}}$. Formulas in $\mathcal{L}_{\mathsf{exp}}$ are built using variables $\{\epsilon\} \cup \{x_i \mid i \in \mathbb{N}\}$, constant symbols $0, 1$, unary relation symbol $e^{(\cdot)}$ applied only to the variable ϵ, binary function symbols $+, -, \times$, and binary relation symbols $=, <$. The terms in the language are integral polynomials with rational coefficients over the variables $\{\epsilon\} \cup \{x_i \mid i \in \mathbb{N}\} \cup \{e^\epsilon\}$. Atomic formulas in the language are of the form $t = 0$ or $t < 0$ or $0 < t$, where t is a term. Quantifier free formulas are Boolean combinations of atomic formulas. Sentences in $\mathcal{L}_{\mathsf{exp}}$ are formulas of the form

$$Q\epsilon Q_1 x_1 \cdots Q_n x_n \psi(\epsilon, x_1, \ldots x_n)$$

where ψ is a quantifier free formula, and Q, Q_is are quantifiers. In other words, sentences are formulas in prenex form, where all variables are quantified, and the outermost quantifier is for the special variable ϵ.

We will be interested in an extension of the first order theory of reals. That is, the theory $\mathsf{Th}_{\mathsf{exp}}$ is the collection of all sentences in $\mathcal{L}_{\mathsf{exp}}$ that hold in the structure $\langle \mathbb{R}, 0, 1, e^{(\cdot)}, +, -, \times, = , < \rangle$, where the interpretation for $0, 1, +, -, \times$ is the standard one on reals, and e is Euler's constant. The crucial property about this theory is that it is decidable.

Theorem 4.2 (McCallum-Weispfenning [13])**.** $\mathsf{Th}_{\mathsf{exp}}$ is decidable.

Our tractable semantic restriction on programs relies on certain special functions of type $I \to \mathbb{R}$, namely those that are definable in $\mathsf{Th}_{\mathsf{exp}}$. A function $f : I \to \mathbb{R}$ is said to be *definable* in $\mathsf{Th}_{\mathsf{exp}}$, if there is a formula $\varphi_f(\epsilon, x)$ in $\mathcal{L}_{\mathsf{exp}}$ with two free variables (ϵ and x) such that

$$f(a) = b \text{ iff } \langle \mathbb{R}, 0, 1, e^{(\cdot)}, +, -, \times, =, < \rangle \models \varphi_f(\epsilon, x)[\epsilon \mapsto a, x \mapsto b]$$

A sufficient condition to ensure the decidability of checking differential privacy is to consider programs with the property that for each input, the probability distribution on the outputs is definable in $\mathsf{Th}_{\mathsf{exp}}$. This identifies the semantic restriction we will consider in this section.

Definition 4.3. A parametrized program P_ϵ with inputs $\mathcal{U}$ and outputs $\mathcal{V}$ is said to identify a *definable distribution* on $\mathcal{V}$ if for each $\mathsf{in} \in \mathcal{U}$ and $\mathsf{out} \in \mathcal{V}$ the function $\epsilon \mapsto \mathsf{Prob}(P_\epsilon(\mathsf{in}) = \mathsf{out})$ is definable in $\mathsf{Th}_{\mathsf{exp}}$.

A parametrized program P_ϵ with inputs $\mathcal{U}$ and outputs $\mathcal{V}$ is said to *effectively* identify a definable distribution on $\mathcal{V}$ if there is an algorithm $\mathcal{A}$ such that for each $\mathsf{in} \in \mathcal{U}$ and $\mathsf{out} \in \mathcal{V}$, $\mathcal{A}$ outputs a formula $\varphi_{\mathsf{in},\mathsf{out}}(\epsilon, x)$ in $\mathcal{L}_{\mathsf{exp}}$ that defines the function $\epsilon \mapsto \mathsf{Prob}(P_\epsilon(\mathsf{in}) = \mathsf{out})$.

The main result of this section is that checking differential privacy for programs that effectively identify a definable distribution is decidable.

Theorem 4.4. The Fixed Parameter Differential Privacy and Differential Privacy problems are decidable for programs P_ϵ that effectively identify a definable distribution and definable functions δ (in the case of the Differential Privacy problem).

Proof Sketch. We sketch the decidability proof for the Differential Privacy problem; the proof also contains all the necessary ideas to establish the decidability of Fixed Parameter Differential Privacy problem. Let P_ϵ be a program that effectively identifies a definable distribution with adjacency relation Φ. Let us assume that the formula $\varphi_{\mathsf{in},\mathsf{out}}(\epsilon, x_{\mathsf{in},\mathsf{out}})$ of $\mathcal{L}_{\mathsf{exp}}$ defines the function $\epsilon \mapsto \mathsf{Prob}(P_\epsilon(\mathsf{in}) = \mathsf{out})$. Let $\varphi_\delta(\epsilon, x_\delta)$ be the formula defining the function δ. Let $t = \frac{p}{q}$ where p, q are natural numbers. We show the proof when I is $(0, \infty)$. It is easy to modify the proof for any interval I with rational end-points.

Consider the sentence

$$\begin{aligned}
\psi = \forall \epsilon. \forall z. [\forall x_{\mathsf{in},\mathsf{out}}]_{\mathsf{in} \in \mathcal{U}, \mathsf{out} \in \mathcal{V}}. \forall x_\delta. \\
((\epsilon > 0) \wedge (e^{p\epsilon} = z^q) \wedge (z > 0) \\
\wedge \varphi_\delta(\epsilon, x_\delta) \ \ \textstyle\bigwedge_{\mathsf{in} \in \mathcal{U}, \mathsf{out} \in \mathcal{V}} \varphi_{\mathsf{in},\mathsf{out}}(\epsilon, x_{\mathsf{in},\mathsf{out}})) \\
\to (\textstyle\bigwedge_{(\mathsf{in}_1, \mathsf{in}_2) \in \Phi, O \subseteq \mathcal{V}} \sum_{\mathsf{out} \in O} x_{\mathsf{in}_1,\mathsf{out}} < \\
z \sum_{\mathsf{out} \in O} x_{\mathsf{in}_2,\mathsf{out}} + x_\delta)
\end{aligned}$$

It is easy to see P_ϵ is differentially private for all ϵ iff ψ is true over the reals. In the syntax of $\mathcal{L}_{\mathsf{exp}}$, we cannot take qth roots of e; therefore, we introduce the variable z, which enables us to write the constraints using only $e^{a\epsilon}$, where $a \in \mathbb{N}$. Notice that ψ belongs to $\mathcal{L}_{\mathsf{exp}}$ if we convert it to prenex form. Decidability therefore follows from the decidability of $\mathsf{Th}_{\mathsf{exp}}$. *q.e.d*

If P_ϵ is not differentially private, then the sentence ψ does not hold. The decision procedure for $\mathsf{Th}_{\mathsf{exp}}$ will in this case return an ϵ that witnesses the non-privacy of P_ϵ. This could be used to construct counterexamples.

Definition 4.5. A counterexample for P_ϵ, with respect to an adjacency relation Φ, a function $\delta : \mathbb{R}^{>0} \to [0, 1]$ and a value $t \in \mathbb{Q}$, is a quadruple (u, u', S, ϵ_0) such that $(u, u') \in \Phi$, $S \subseteq \mathcal{V}$ and $\epsilon_0 > 0$ and

$$\mathsf{Prob}(P_{\epsilon_0}(u) \in S) > e^{t\epsilon_0}\, \mathsf{Prob}(P(u') \in S) + \delta(\epsilon_0)$$

When δ is the constant function 0, then S is $\{v\}$ for some $v \in \mathcal{V}$.

CHAPTER 5: DIPWHILE LANGUAGE

We now introduce the language DiPWhile for which differential privacy can be checked effectively (Chapter 6). Informally, DiPWhile is a syntactically restricted class of probabilistic while programs, having variables that take values from either Booleans, a finite set DOM (to model variables taking finite values), integers, or reals. Probabilistic steps in the language correspond to sampling using either the Laplace, discrete Laplace, or exponential mechanisms. We also allow probabilistic steps where values in DOM are sampled from a user defined distribution. The key restrictions we impose are as follows. First, we assume that real and integer variables are never assigned inside the scope of a while loop. This ensures that any real (or integer) variable is given a value only a *bounded* number of times. Second, loop and branch conditionals never depend on comparing values stored in real variables with values stored in integer variables. These restrictions are crucially exploited in our decidability proof. The informal reasons behind them can be best understood in the context of defining the semantics for DiPWhile programs and so are postponed to Section 6.2.

The formal syntax of DiPWhile, that makes precise the restrictions outlined above, is shown in Figure 5.1. We have four types for variables: $Bool = \{\mathsf{true}, \mathsf{false}\}$; finite domain DOM that we assume (without loss of generality) to be $\{-N_{\mathsf{max}}, \ldots 0, 1, \ldots N_{\mathsf{max}}\}$, a finite subset of integers [1]; reals $\mathbb{R}$; and integers $\mathbb{Z}$. The set of Boolean/DOM/integer/real program variables are respectively denoted by $\mathcal{B}/\mathcal{X}/\mathcal{Z}/\mathcal{R}$. The set of Boolean/DOM/integer/real expressions is given by the non-terminal $B/E/Z/R$ in Figure 5.1. We now explain the rules for such expressions. Boolean expressions (B) can be built using Boolean variables and constants, standard Boolean operations, and by applying functions from $\mathcal{F}_{Bool}$. $\mathcal{F}_{Bool}$ is assumed to be a collection of *computable* functions returning a $Bool$. DOM expressions (E) are similarly built from DOM variables, values in DOM, and applying functions from set of computable functions $\mathcal{F}_{\mathsf{DOM}}$. Next, integer expressions (Z) are built using multiplication and addition with integer constants and DOM expressions, and additions with other integer expressions. Finally, real expressions (R) are built using multiplication and addition with rational constants and DOM expressions, and additions with other real-valued expressions.

A program in our language is a triple consisting of a set of (private) input variables, a set of (public) output variables, and a finite sequence of labeled statements (non-terminal P in Figure 5.1). The private input variables and public output variables take values from the domain DOM. Thus, the set of possibles inputs/outputs ($\mathcal{U}/\mathcal{V}$), is identified with the set of valuations for input/output variables; a valuation over a set of variables $X' = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_m\} \subseteq \mathcal{X}$ is a

[1] Our decidability results also hold if DOM is taken to be a finite subset of the rationals.

Expressions ($\mathsf{b} \in \mathcal{B}, \mathsf{x} \in \mathcal{X}, \mathsf{z} \in \mathcal{Z}, \mathsf{r} \in \mathcal{R}, d \in \mathsf{DOM}, i \in \mathbb{Z}, q \in \mathbb{Q}, g \in \mathcal{F}_{Bool}, f \in \mathcal{F}_{\mathsf{DOM}}$):

$$\begin{array}{rcl} B & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{b} \mid not(B) \mid B \ and \ B \mid B \ or \ B \mid g(\tilde{E}) \\ E & ::= & d \mid \mathsf{x} \mid f(\tilde{E}) \\ Z & ::= & \mathsf{z} \mid iZ \mid EZ \mid Z + Z \mid Z + i \mid Z + E \\ R & ::= & \mathsf{r} \mid qR \mid ER \mid R + R \mid R + q \mid R + E \end{array}$$

Basic Program Statements ($a \in \mathbb{Q}^{>0}$, $\sim \in \{<, >, =, \leq, \geq\}$, F is a scoring function and choose is a user-defined distribution):

$$\begin{array}{rcl} s & ::= & \mathsf{x} \leftarrow E \mid \mathsf{z} \leftarrow Z \mid \mathsf{r} \leftarrow R \mid \mathsf{b} \leftarrow B \mid \mathsf{b} \leftarrow Z_1 \sim Z_2 \mid \\ & & \mathsf{b} \leftarrow Z \sim E \mid \mathsf{b} \leftarrow R_1 \sim R_2 \mid \mathsf{b} \leftarrow R \sim E \mid \\ & & \mathsf{r} \leftarrow \mathsf{Lap}(a\epsilon, E) \mid \mathsf{z} \leftarrow \mathsf{DLap}(a\epsilon, E) \mid \\ & & \mathsf{x} \leftarrow \mathsf{Exp}(a\epsilon, F(\tilde{\mathsf{x}}), E) \mid \mathsf{x} \leftarrow \mathsf{choose}(a\epsilon, \tilde{E}) \mid \\ & & \mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ P\ \mathsf{end} \mid \mathsf{While}\ B\ \mathsf{do}\ P\ \mathsf{end} \mid \mathsf{exit} \end{array}$$

Program Statements ($\ell \in \mathsf{Labels}$)

$$P \ ::= \ \ell : \ s \mid \ell : \ s ; \ P$$

Figure 5.1: BNF grammar for DiPWhile. DOM is a finite discrete domain. $\mathcal{F}_{Bool}$, ($\mathcal{F}_{\mathsf{DOM}}$ resp) are set of functions that output Boolean values (DOM respectively). $\mathcal{B}, \mathcal{X}, \mathcal{Z}, \mathcal{R}$ are the sets of Boolean variables, DOM variables, integer random variables and real random variables. Labels is a set of program labels. For a syntactic class S, $\tilde{S}$ denotes a sequence of elements from S.

function from X' to DOM. Note that if we represent the set X' as a sequence $\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_m$ then we can represent a valuation val over x as a sequence $val(\mathsf{x}_1), val(\mathsf{x}_2), \ldots, val(\mathsf{x}_m)$ of elements from DOM.

We assume every statement in our program is uniquely labeled from a set of called Labels. Statements (non-terminal s) can either be assignments, conditionals, while loops, or exit. Statements other than assignments are self-explanatory. The syntax of assignments is designed to follow a strict discipline. Real and integer variables can either be assigned the value of real/integer expression or samples drawn using the Laplace or discrete Laplace mechanism. DOM variables are either assigned values of DOM expressions or values drawn either using an exponential mechanism ($\mathsf{Exp}(a\epsilon, F(\tilde{\mathsf{x}}), E)$) or a user defined distribution ($\mathsf{choose}(a\epsilon, \tilde{E})$). For the exponential mechanism, we require that the scoring function F be computable and return a rational value. Both these restrictions are unlikely to be severe in practice, but are needed to ensure decidability. In the case of the user defined distribution, we demand that

the probability with which a value in DOM is chosen (as function of the privacy budget ϵ), be definable in $\mathsf{Th}_{\mathsf{exp}}$. Again this is needed to ensure decidability. Finally, we consider assignments to Boolean variables. The interesting cases are those where the Boolean variable stores the result of the comparison of two expressions. As mentioned at the beginning of this section, we do not allow that comparison between real and integer expressions; this is reflected in the syntax. In addition to the above syntactic restrictions, DiPWhile programs will adhere to the following principles.

Bounded Assignments We do not allow assignments to real and integer variables within the scope of a while loop. This ensures that assignments to such variables happen only a *bounded* number of times during an execution. Therefore, without loss of generality, we will assume that real and integer variables are assigned *at most once*.

Define Before Use We will assume that in any execution, if a variable appears on the right side of an assignment statement, then it should have been assigned a value before.

The DiPWhile language is surprisingly expressive — many known randomized algorithms for differential privacy can be encoded. We give examples of such encodings in DiPWhile. We omit labels of program statements unless they are needed.

Example 5.1. Figure 5.1 shows how SVT can be encoded in our language with $T = 0, \Delta = 1, N = 2, c = 1$. In the example we are modeling $\perp$ by 0 and $\top$ by 1. Please observe that although we do not have For loops in our program, we can nevertheless encode bounded For Loops by unrolling the For loop.

Example 5.2. Given $\epsilon > 0$ and offset, let $\mathsf{Lap}^+(\epsilon, \mathsf{offset})$ be the continuous distribution whose probability density function (p.d.f.) is given by

$$f_{\epsilon,\mu}(x) = \begin{cases} \epsilon\ e^{-\epsilon(x-\mathsf{offset})} & \text{if } x \geq \mathsf{offset} \\ 0 & \text{otherwise} \end{cases}$$

Observe that the one-sided Laplacian distribution $\mathsf{Lap}^+(\epsilon, 0)$ is the standard exponential distribution. Our language is expressive enough to encode one-sided Laplacians as follows. Consider the sequence of statements:

$$\begin{array}{l} X \leftarrow \mathsf{Lap}(\epsilon, 0); \\ b \leftarrow X \leq 0; \\ \mathsf{if}\, b\, \mathsf{then}\, Y \leftarrow X\, \mathsf{else}\, Y \leftarrow (-1)X\, \mathsf{end}; \\ Z \leftarrow Y + \mathsf{offset} \end{array}$$

Input: q_1, q_2
Output: out_1, out_2

$T \leftarrow 0$;
$out_1 \leftarrow 0$;
$out_2 \leftarrow 0$;
$\mathsf{r}_T \leftarrow \mathsf{Lap}(\frac{\epsilon}{2}, T)$;
$\mathsf{r}_1 = \mathsf{Lap}(\frac{\epsilon}{4}, q_1)$;
$\mathsf{b} \leftarrow \mathsf{r}_1 \geq \mathsf{r}_T$;
if b then
  $out_1 \leftarrow 1$
else
  $\mathsf{r}_2 = \mathsf{Lap}(\frac{\epsilon}{4}, q_2)$;
  $\mathsf{b} \leftarrow \mathsf{r}_2 \geq \mathsf{r}_T$;
  if b then
    $out_2 \leftarrow 1$
  end
end
exit

Algorithm 5.1: SVT for 1-sensitive queries with $N = 2$,$c = 1$ and $T = 0$

The effect of the sequence of statements is that Z has the one-sided Laplacian distribution $\mathsf{Lap}^+(\epsilon, \mathsf{offset})$.

Other examples that can be encoded in our language (and for which the decision procedure applies) include randomized response, the multiplicate weights and iterative database construction [21, 22], the private smart sum algorithm [23], and private vertex cover [24].

CHAPTER 6: DECIDABILITY OF DIPWHILE PROGRAMS

We will now prove the main result of this report — the decidability of the Fixed Parameter Differential Privacy and Differential Privacy problems for DiPWhile programs. Our proof rests on two observations. First, the semantics of DiPWhile programs can be defined as finite state discrete time Markov chains (DTMC). This observation is surprising because DiPWhile programs have real and integer values variables, and so a naïve definition of semantics will have infinitely many states. The key insight in establishing this observation is that a precise semantics of DiPWhile programs can be defined without explicitly tracking the values of real and integer-valued variables. Second, all the transition probabilities arising in our semantics are definable in $\mathsf{Th}_{\mathsf{exp}}$. These two observations allow us to use Theorem 4.4 to establish decidability of checking differential privacy of DiPWhile programs.

The above proof outline motivates the organization of this section. We begin by introducing parametrized DTMCs that are used to define the semantics of DiPWhile programs (Section 6.1). Next (Section 6.2), we define the semantics of DiPWhile programs using finite state parametrized DTMCs.

6.1 PARAMETRIZED DTMCS

Discrete time Markov chains (DTMC) are transition systems where transitions between states are the result of a coin toss, as opposed to a nondeterministic choice. DiPWhile programs depend on the privacy budget ϵ, and the distributions used to sample random values may depend on ϵ. Therefore, our semantics for programs will yield a DTMC whose transition probabilities depend on ϵ. This leads us to the notion of a *parametrized DTMC* that we define below.

Definition 6.1. A *parametrized DTMC* is a tuple $\mathcal{D}_\epsilon = (Z, \Delta)$, where Z is a (countable) set of states, and $\Delta : Z \times Z \to (\mathbb{R}^{>0} \to [0,1])$ is the *probabilistic transition function*. For any pair of states z, z', Δ returns a function from $\mathbb{R}^{>0}$ to $[0,1]$, such that for every $\epsilon > 0$, $\sum_{z' \in Z} \Delta(z,z')(\epsilon) = 1$. We will call $\Delta(z,z')$ as the probability of transitioning from z to z'.

A *definable* parametrized DTMC is a parametrized DTMC $\mathcal{D}_\epsilon = (Z, \Delta)$ such that for every pair of states $z, z' \in Z$, the function $\Delta(z,z')$ is definable in $\mathsf{Th}_{\mathsf{exp}}$.

In this report we will primarily be interested in parametrized DTMCs that have finitely many states and which are definable. A parametrized DTMC associates with each (finite) sequence of states $\rho = z_0, z_1, \ldots z_m$, a function $\mathsf{Prob}(\rho) : \mathbb{R}^{>0} \to [0,1]$ that given an $\epsilon > 0$,

returns the probability of the sequence ρ when the parameter's value is fixed to ϵ, i.e.,

$$\mathsf{Prob}(\rho)(\epsilon) = \prod_{i=0}^{m-1} \Delta(z_i, z_{i+1})(\epsilon).$$

For a state z_0 and a set of states $Z' \subseteq Z$, once again we have a function that given a value ϵ for the parameter, returns the probability of reaching Z' from z_0. This can be formally defined as

$$\mathsf{Prob}(z_0, Z')(\epsilon) = \sum_{\rho \in z_0(Z \setminus Z')^* Z'} \mathsf{Prob}(\rho)(\epsilon).$$

In other words, $\mathsf{Prob}(z_0, Z')(\epsilon)$ is the sum of the probability of all sequences starting in z_0, ending in Z', such no state except the last is in Z'. We end the section with an important observation about finite, definable, parametrized DTMCs.

Theorem 6.2. For any finite state, definable, parametrized DTMC $\mathcal{D}_\epsilon$, any state z_0 and set of states Z', the function $\mathsf{Prob}(z_0, Z')$ is definable in $\mathsf{Th}_{\mathsf{exp}}$. Moreover, there is an algorithm that computes the formula defining $\mathsf{Prob}(z_0, Z')$.

Proof. Let us first recall how reachability probabilities are computed in (non-parametrized) finite state DTMCs. Recall that a (non-parametrized) DTMC is a pair (Q, δ) where Q is a finite set of states, and $\delta : Q \times Q \to [0, 1]$ is such that for every $q \in Q$, $\sum_{q' \in Q} \delta(q, q') = 1$. So in a DTMC the transition probabilities are fixed, and are not functions of a parameter. The probability of reaching a set of states $Q' \subseteq Q$ from a state q_0 is computed by solving a more general problem, namely, the problem of computing the probability of reaching Q' from each state $q \in Q$. Let the variable x_q denote the probability of reaching Q' from state q. One simple observation is that if $q \in Q'$ then $x_q = 1$. Second, if Q_0 denotes the set of all states from which Q' is not reachable in the underlying graph (i.e., one where we ignore the probabilities and just have edges for all transitions that are non-zero), then $x_q = 0$ if $q \in Q_0$. Now the set Q_0 can be computed by performing a simple graph search on the underlying graph. For states $q \notin (Q' \cup Q_0)$, we could write x_q as $x_q = \sum_{q' \in Q} \delta(q, q') x_{q'}$. This gives us the following system of linear equations.

$$\begin{array}{ll} x_q = 1 & \text{if } q \in Q' \\ x_q = 0 & \text{if } q \in Q_0 \\ x_q = \sum_{q' \in Q} \delta(q, q') x_{q'} & \text{otherwise} \end{array}$$

The above system of linear equations can be shown to have a unique solution, with the solution giving the probability of reaching Q' from each state q.

Now let us consider a parametrized DTMC $\mathcal{D} = (Z, \Delta)$. Let $\varphi_{zz'}$ be a $\mathcal{L}_{\mathsf{exp}}$ formula that defines the function $\Delta(z, z')$. Recall that in the algorithm outlined in the previous paragraph, one crucial step is to compute the set of states that have probability 0 of reaching the target set. This requires knowing the underlying graph of the DTMC, i.e., knowing which transitions have probability 0 and which ones have probability > 0. In a parametrized DTMC this is challenging because the probability of transitions depends on the value of ϵ, and our goal is to compute the reachability probability as a function of ϵ. We will overcome this challenge by "guessing" the underlying graph.

Let $C \subseteq Z \times Z$. We will construct a formula φ_C that will capture the constraints that reachablity probabilities need to satisfy under the assumption that the probability of edges in C is 0, and those outside C is > 0. Based on the assumption that C is exactly the set of 0 probability edges, we can compute the set Z_0^C of states that cannot reach Z'. The formula φ_C will have variables that will have the following intuitive interpretations — $p_{zz'}$ the probability of transitioning from z to z'; x_z the probability of reaching Z' from state z.

$$\begin{aligned}\varphi_C = \textstyle\bigwedge_{(z,z')\in C}(p_{zz'} = 0) \wedge \bigwedge_{(z,z')\notin C}(p_{zz'} > 0) \wedge \bigwedge_{z\in Z'}(x_z = 1)\\ \wedge \textstyle\bigwedge_{z\in Z_0^C}(x_z = 0) \wedge \bigwedge_{z\notin(Z'\cup Z_0^C)}(x_z = \sum_{z'} p_{zz'}x_{z'}).\end{aligned}$$

Notice that φ_C is a formula in $\mathcal{L}_{\mathsf{exp}}$. φ_C can be used to construct the formula we want. To construct the formula $\varphi_{z_0Z'}$ that characterizes the probability of reaching Z' from z_0, we need to account for two things. First, we need to ensure that $p_{zz'}$ is indeed the probability of transitioning from z to z'. Second, we need to account for the fact that we don't know the exact set of edges with probability 0. Based on these observations, we can define $\varphi_{z_0,Z'}$ as follows.

$$\varphi_{z_0Z'} = [\exists x_z]_{z\neq z_0}[\exists p_{zz'}]_{z,z'\in Z} \bigwedge_{z,z'\in Z} \varphi_{zz'}(\epsilon, p_{zz'}) \wedge \left(\bigvee_{C\subseteq Z\times Z} \varphi_C\right)$$

In the above definition of $\varphi_{z_0Z'}$ all variables except x_{z_0} (and ϵ) are existentially quantified. Notice, that $\varphi_{z_0Z'}$ is in $\mathcal{L}_{\mathsf{exp}}$ provided we pull all the quantifiers to get it in prenex form. Given that Z_0^C can be effectively constructed for any set C, the above formula can also be computed for any parametrized DTMC $\mathcal{D}$. *q.e.d*

6.2 SEMANTICS

We now sketch the challenges in defining the semantics of DiPWhile programs, and our key insights in overcoming them. Let us fix a program P_ϵ. A naïve definition of the semantics, $[\![P_\epsilon]\!]$, of P_ϵ would have as states the label of the statement of P_ϵ to be executed next, along

with a valuation that assigns to each program variable the values currently stored in them. The problem is, since P_ϵ has real and integer valued variables, such a semantics will have uncountably many states. Defining the probability of executions becomes mathematically involved and it is unclear how to design decision procedures for it.

Our key insight in defining $[\![P_\epsilon]\!]$ as a finite state, parametrized DTMC, is that we do not need to track the values of real and integer valued variables. Our state is going to be a tuple of the form $(\ell, f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C)$ where ℓ is the label of the statement of P_ϵ to be executed next. The functions f_{Bool} and f_{DOM} assign values to the *Bool* and DOM variables, respectively; this is just like in the naïve semantics. Let us now look at f_{real}. Intuitively, f_{real} is supposed to be the "valuation" for the real variables. But instead of mapping each variable to a value in $\mathbb{R}$, we will instead map it to a finite set. To understand this mapping, let us recall that in DiPWhile a real variable is assigned only once in a program. Further, such an assignment either assigns the value of a linear expression over program variables, or samples using a Laplace mechanism. Therefore, f_{real} will map a variable to either the linear expression it is assigned, or the expressions defining the parameters of the Laplace mechanism used in sampling. Notice that the range of f_{real} is now a finite set. Similarly, f_{int} maps each integer variable to either the linear expression it is assigned or the parameters of the discrete Laplace mechanism. The last state component C is the set of Boolean conditions on real and integer variables that hold along the path thus far; this will become clearer when we describe the transitions. Since the Boolean conditions must be Boolean expressions in the program or their negation, C is also a finite set. These observations show that we will have finitely many states.

We now sketch how the state is updated in $[\![P_\epsilon]\!]$. Updates to DOM variables will be as expected — it will be a probabilistic transition if the assignment samples using an exponential mechanism or a user defined distribution, and it will be a deterministic step updating f_{DOM} otherwise. Assignments to real variables are always deterministic steps that change the function f_{real}. Thus, even if the step samples using the Laplace mechanism, in the semantics it will be modeled as a deterministic step where f_{real} is updated by storing the parameters of the distribution. Similarly all integer assignments are deterministic steps as well. Steps where a Boolean variable is assigned a Boolean expressions will be modeled as expected — we update the valuation f_{Bool} to reflect the assignment. The interesting case is, $\mathsf{b} \leftarrow R_1 \sim R_2$, when a boolean variable gets assigned the result of the comparison of two real expressions; the case of comparing two integer expressions is similar. In this case, we will transition to a state where $R_1 \sim R_2$ is added to C with probability equal to the probability that $(R_1 \sim R_2)$ holds conditioned on the fact that C holds; if the probability of C holding is 0 then the DTMC will transition to a special reject state. With the remaining probability we

will transition to the state where $\neg(R_1 \sim R_2)$ is added to C. Thus, Boolean assignments will be modeled by probabilistic transitions. Finally, branches and while loop conditions are modeled as deterministic steps, with choice of the next statement being determined by the value of the Boolean variable (of the condition) in f_{Bool}. These informal ideas are fleshed out in the Section 6.2.1 to give a precise mathematical definition.

It is worth noting how key syntactic restrictions in DiPWhile programs play a role in defining its semantics. The first restriction is that integer and real variables are not assigned in the scope of a while loop. This is critical to ensure that the DTMC $[\![P_\epsilon]\!]$ is finite state. Since we track distribution parameters and linear expressions for such variables, this restriction ensures that we only remember a bounded number of these. Second, DiPWhile disallows comparison between real and integer expressions in its syntax. Recall that such comparison steps result in a probabilistic transition, where we compute the probability of the comparison holding conditioned on the properties in C holding. It is unclear how to compute these probabilities for comparisons between integer and real random variables. Hence they are disallowed.

6.2.1 Formal Semantics of DiPWhile programs

Let us recall some key restrictions in DiPWhile programs. The first restriction is that real and integer-valued variables are never assigned within the scope of a while statement. Hence, they are assigned only a bounded number of times, and therefore, without loss of generality, we can assume that they are assigned a value exactly *once*. Second, real valued expressions are never compared against integer valued expressions.

Let us fix some basic notation. Partial functions from A to B will be denoted as $A \hookrightarrow B$. The value of $f : A \hookrightarrow B$ on $a \in A$, will be denoted as $f(a)$. Two partial functions f and g will be equal (denoted $f \simeq g$) if for every element a, either f and g are both undefined, or $f(a) = f(b)$. If $f : A \hookrightarrow B$, $a \in A$ and $b \in B$, then $f[a \mapsto b]$ denotes the partial function that agrees with f on all elements of A except a; on a, $f[a \mapsto b](a) = b$.

In the rest of this section let us fix a DiPWhile program P_ϵ. L will denote the set of labels appearing in P_ϵ. A valuation val for DOM variables is a function that assigns a value in DOM to variables in $\mathcal{X}$; we will denote set of all such valuations by $\mathsf{V}_{\mathsf{DOM}}$. Given a valuation $val \in \mathsf{V}_{\mathsf{DOM}}$ and a real expression e, $val(e)$ denotes the real expression that results from substituting all the DOM variables appearing in e by their value in val. Similarly, for an integer expression, $val(e)$ is the partial evaluation of e with respect to val. Finally, for a comparison $e_1 \sim e_2$ between two expressions e_1 and e_2, again we will define $val(e_1 \sim e_2)$ to be $val(e_1) \sim val(e_2)$. Let us denote the set of integer expressions, real expressions, and

Boolean comparisons, appearing on the right hand side of assignments in P_ϵ by P_Z, P_R, and P_B, respectively. Three sets of expressions will be used in defining the semantics, and they are as follows.

$$\mathsf{zExp} = \{val(e) \mid val \in \mathsf{V_{DOM}},\ e \in P_Z\}$$
$$\mathsf{rExp} = \{val(e) \mid val \in \mathsf{V_{DOM}},\ e \in P_R\}$$
$$\mathsf{bExp} = \{val(e) \mid val \in \mathsf{V_{DOM}},\ e \in P_B\}$$

Thus, zExp, rExp, and bExp are partially evaluated expression appearing on the right hand side of assignments in P_ϵ. Notice that the sets L, zExp, rExp, and bExp are all finite. Finally, let Const be the set of rational constants appearing as coefficient of ϵ of Laplace and discrete Laplace assignments in P_ϵ; again Const is finite.

In order to define the semantics of P_ϵ, we will use an auxiliary function next that given a label, identifies the label of the statement to be executed next. Observe that for most program statements, the next statement to be executed is unique. However, for if and While statements, the next statement depends on the value of a Boolean expression. We will define $\mathsf{next}(\ell)$ to be a set of pairs of the form (ℓ', c) with the understanding that ℓ' is the next label if c holds. Thus, for a label ℓ, $\mathsf{next}(\ell)$ will either be $\{(\ell', \mathsf{true})\}$ or $\{(\ell_1, c), (\ell_2, \neg c)\}$. We do not give a precise definition of $\mathsf{next}(\cdot)$, but we will use it when defining the semantics.

The semantics of P_ϵ will given as a finite state, parametrized DTMC $[\![P_\epsilon]\!]$. To define the parametrized DTMC $[\![P_\epsilon]\!]$, we need to define the states and the transitions.

States States of $[\![P_\epsilon]\!]$ will be of the form

$$(\ell, f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C).$$

Informally, $\ell \in \mathsf{L}$ is the label of the statement to be executed, f_{Bool}, f_{DOM}, f_{int}, and f_{real} are partial functions assigning "values" to program variables (of appropriate type), and C is a collection of inequalities among program variables that hold on the current computational path. Both f_{Bool} and f_{DOM} are valuations for the appropriate set of variables, and so we have $f_{Bool} : \mathcal{B} \hookrightarrow \{\mathsf{true}, \mathsf{false}\}$ and $f_{\mathsf{DOM}} : \mathcal{X} \hookrightarrow \mathsf{DOM}$. For real and integer variables, instead of tracking exact values, we will track the expressions used in assignments and parameters of (discrete) Laplace mechanisms used in random assignments. Therefore, we have $f_{\mathsf{int}} : \mathcal{Z} \hookrightarrow \mathsf{zExp} \cup (\mathsf{Const} \times \mathsf{DOM})$ and $f_{\mathsf{real}} : \mathcal{R} \hookrightarrow \mathsf{rExp} \cup (\mathsf{Const} \times \mathsf{DOM})$. Finally, $C \subseteq \mathsf{bExp} \cup \{\neg e \mid e \in \mathsf{bExp}\}$. It follows immediately that the set of states of $[\![P_\epsilon]\!]$ is finite.

Well Formed States The functions f_* (for $* \in \{Bool, \mathsf{DOM}, \mathsf{int}, \mathsf{real}\}$) assign values to program variables that have been assigned during the computation thus far. Since we assume

variables in DiPWhile program are defined before they are used, if a variable z' appears in $f_{\mathsf{int}}(\mathsf{z}) \in \mathsf{zExp}$, then $f_{\mathsf{int}}(\mathsf{z}')$ must be defined. A similar condition holds for real variables. The comparisons in C are also relationships that must hold on the current path, and so all variables participating in it must be defined. If a state satisfies these consistency properties between f_{int}, f_{real}, and C, we will say it is *well formed*. All reachable states in $[\![P_\epsilon]\!]$ will be well formed. So when we define transitions we will assume that the states are well formed.

Initial States Let ℓ_{in} be the label of the first statement P_ϵ. Let $C^{\mathsf{in}} = \emptyset$, and let f^{in}_{Bool}, $f^{\mathsf{in}}_{\mathsf{int}}$, and $f^{\mathsf{in}}_{\mathsf{real}}$ be partial functions with an empty domain. An initial state of $[\![P_\epsilon]\!]$ will be of the form $(\ell_{\mathsf{in}}, f^{\mathsf{in}}_{Bool}, f^{\mathsf{in}}_{\mathsf{DOM}}, f^{\mathsf{in}}_{\mathsf{int}}, f^{\mathsf{in}}_{\mathsf{real}}, C^{\mathsf{in}})$, where $f^{\mathsf{in}}_{\mathsf{DOM}}$ is defined only on the input variables; the values given to these variables by $f^{\mathsf{in}}_{\mathsf{DOM}}$ will be the "initial input value".

We will now define the semantics of transitions in $[\![P_\epsilon]\!]$. For this, let us fix a state $z = (\ell, f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C)$. Transitions out of z will be defined based on describe the effect of executing the statement labeled ℓ, and so its definition will depend on this statement. We handle each case below.

DOM assignments Let $\mathsf{next}(\ell) = \{(\ell', \mathsf{true})\}$ and let x be the variable being assigned in ℓ. There are two cases to consider. First, consider the case where x is assigned a value for a DOM expression e. In this case, $[\![P_\epsilon]\!]$ will transition to

$$(\ell', f_{Bool}, f_{\mathsf{DOM}}[\mathsf{x} \mapsto f_{\mathsf{DOM}}(e)], f_{\mathsf{int}}, f_{\mathsf{real}}, C)$$

with probability 1. The second case is when x is assigned a random value according to $\mathsf{Exp}(a\epsilon, F(\tilde{\mathsf{x}}), e)$ or $\mathsf{choose}(a\epsilon, \tilde{e})$. For $d \in \mathsf{DOM}$, let $\mathsf{prob}(d)$ be the probability of d (as a function of ϵ) based on the distribution; note, that these probabilities will depend on the value of $f_{\mathsf{DOM}}(e)$ and $f_{\mathsf{DOM}}(\tilde{e})$. Then, $[\![P_\epsilon]\!]$ will transition to

$$(\ell', f_{Bool}, f_{\mathsf{DOM}}[\mathsf{x} \mapsto d], f_{\mathsf{int}}, f_{\mathsf{real}}, C)$$

with probability $\mathsf{prob}(d)$.

Integer assignments Let $\mathsf{next}(\ell) = \{(\ell', \mathsf{true})\}$ and let z be the variable being assigned in ℓ. Again there are two cases to consider. First, consider the case where z is assigned a value for an integer expression e. In this case, $[\![P_\epsilon]\!]$ will transition to

$$(\ell', f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}[\mathsf{z} \mapsto f_{\mathsf{DOM}}(e)], f_{\mathsf{real}}, C)$$

with probability 1. Next, if z is assigned a random value according to $\mathsf{DLap}(a\epsilon, e)$, then $[\![P_\epsilon]\!]$ transitions to

$$(\ell', f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}[\mathsf{z} \mapsto (a, f_{\mathsf{DOM}}(e))], f_{\mathsf{real}}, C)$$

with probability 1. Notice that we have a deterministic transition even if the assignment samples from a discrete Laplace. The effect of choosing randomly a value will get accounted for during Boolean assignments.

Real assignments Let $\mathsf{next}(\ell) = \{(\ell', \mathsf{true})\}$ and let r be the variable being assigned in ℓ. First, if z is assigned a value for a real expression e, $[\![P_\epsilon]\!]$ will transition to

$$(\ell', f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}[\mathsf{r} \mapsto f_{\mathsf{DOM}}(e)], C)$$

with probability 1. If z is assigned a random value according to $\mathsf{Lap}(a\epsilon, e)$, then $[\![P_\epsilon]\!]$ transitions to

$$(\ell', f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}[\mathsf{r} \mapsto (a, f_{\mathsf{DOM}}(e))], C)$$

with probability 1. Again sampling according to Laplace is modeled deterministically.

Boolean assignments Again let $\mathsf{next}(\ell) = \{(\ell', \mathsf{true})\}$ and let b be the variable being assigned in ℓ. When b is assigned the value of Boolean expression e, $[\![P_\epsilon]\!]$ transitions to

$$(\ell', f_{Bool}[\mathsf{b} \mapsto f_{Bool}(e)], f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C)$$

with probability 1. The interesting case is when b is assigned the result of comparing expressions $e_1 \sim e_2$. Let p_1 denote the probability of $f_{\mathsf{DOM}}(e_1) \sim f_{\mathsf{DOM}}(e_2)$ holding given all conditions in C hold; notice that this probability depends on the functions f_{int} and f_{real} that store the parameters to various random sampling steps. Now $[\![P_\epsilon]\!]$ will transition to

$$(\ell', f_{Bool}[\mathsf{b} \mapsto \mathsf{true}], f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C \cup \{f_{\mathsf{DOM}}(e_1) \sim f_{\mathsf{DOM}}(e_2)\})$$

with probability p_1, and it will transition to

$$(\ell', f_{Bool}[\mathsf{b} \mapsto \mathsf{false}], f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C \cup \{\neg(f_{\mathsf{DOM}}(e_1) \sim f_{\mathsf{DOM}}(e_2))\})$$

with probability $1 - p_1$. The effect of the probabilistic sampling steps for integer and real variables gets accounted for when the result of a comparison is assigned to a Boolean variable.

if statement In this case, $\mathsf{next}(\ell) = \{(\ell_1, c), (\ell_2, \neg c)\}$. If $f_{Bool}(c) = \mathsf{true}$ then we transition to

$$(\ell_1, f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C)$$

with probability 1. On the other hand, if $f_{Bool}(c) = \mathsf{false}$ then transition to

$$(\ell_2, f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C)$$

with probability 1.

While statement Again let $\mathsf{next}(\ell) = \{(\ell_1, c), (\ell_2, \neg c)\}$. This case is identical to the case of if statement, and so is skipped.

exit statement In this case we stay in state z with probability 1.

6.2.2 DiPWhile Programs are finite, definable, parametrized DTMCs

Probabilistic transitions in our semantics arise due to two reasons. First are assignments to DOM variables that sample according to either the exponential or a user defined distribution. Our restrictions on exponential mechanism (that scoring functions take rational values) and on user defined distributions, ensure that the resulting probabilities in these transitions can be defined in $\mathsf{Th}_{\mathsf{exp}}$. The second is due to comparisons between expressions. We can prove that in this case also, the resulting probabilities are definable in $\mathsf{Th}_{\mathsf{exp}}$.

Theorem 6.3. For any DiPWhile program P_ϵ, $[\![P_\epsilon]\!]$ is a finite, definable, parametrized DTMC that is computable.

Proof. From our definition of the semantics (Section 6.2.1), it follows that $[\![P_\epsilon]\!]$ is a finite parameterized DTMC. We now show that it is definable also. In order to show this, we have to show that the transition probabilities of $[\![P_\epsilon]\!]$ are definable. Observe that, by definition, the transition probabilities of $\mathsf{choose}(a\epsilon, \tilde{E})$ construct are definable. The other probabilistic transitions arise as a result of comparison between random variables of the same sort or from using the exponential mechanism. These transition probabilities turn out to be from a special class of definable functions. We define this form next.

Definition 6.4. Let $p(\epsilon) = \sum_{i=1}^{m} a_i \epsilon^{n_i} e^{\epsilon q_i}$ where each a_i is a rational number, n_i is a natural number and q_i is a non-negative rational number. We shall call all such expressions *pseudo-polynomials* in ϵ. Given a real number $b > 0$ and a pseudo-polynomial $p(\epsilon)$, $p(b)$ is the real number obtained by substituting b for ϵ. The ratio of two pseudo-polynomials in ϵ, $\frac{p_1(\epsilon)}{p_2(\epsilon)}$,

shall be called a *pseudo-rational function* in ϵ if $p_2(b) \neq 0$ for all real $b > 0$. Given a real number $b > 0$ and a pseudo-rational function $rt(\epsilon) = \frac{p_1(\epsilon)}{p_2(\epsilon)}$, $rt(b)$ is defined to be $\frac{p_1(b)}{p_2(b)}$.

Observe that a pseudo-rational function rt defines a function f_{rt} from the set of strictly positive reals to the set of reals. We will henceforth confuse f_{rt} with rt. Pseudo-rational functions are easily seen to be closed under addition and multiplication.

Proposition 6.5. Each pseudo-rational function rt is definable in the theory $\mathsf{Th}_{\mathsf{exp}}$.

Proof. Let $rt(\epsilon) = \frac{\sum_{i=1}^{m} a_i \epsilon^{n_i} e^{\epsilon q_i}}{\sum_{i=1}^{m'} a'_i \epsilon^{n'_i} e^{\epsilon q'_i}}$. Let N be the least common multiple of all denominators of q_i, q'_i. Let $p_i = q_i N$ and $p'_i = q'_i N$. It is easy to see that rt is definable by the formula

$$\forall z.((x \sum_{i=1}^{m'} a'_i \epsilon^{n'_i} z^{p'_i} = \sum_{i=1}^{m} a_i \epsilon^{n_i} z^{p_i}) \wedge (z^N = e^\epsilon) \wedge (z > 0)).$$

Note that in the above formula, z is the Nth root of ϵ. *q.e.d*

Now, it follows from our restriction on our scoring functions, namely that they take values in rationals, that the transition probabilities in exponential mechanism are pseudo-rational functions that can be computed.

Let us now consider the case of comparison between random variables. Let $state = (\ell, f_{Bool}, f_{\mathsf{DOM}}, f_{\mathsf{int}}, f_{\mathsf{real}}, C)$ of $[\![P_\epsilon]\!]$ be a state of $[\![P_\epsilon]\!]$. Recall that when we compare random variables in *state*, we add a new linear comparison e to C. Further, in order to compute transition probabilities, we compute the conditional probability that the set of linear comparison $C \cup e$ is true given that C is true. For this, it suffices to show that we can compute the probability that the set of linear comparisons C is true and the probability $C \cup e$ is true. We make the following observations:

- Since every random variable must be defined before it is used, we can simplify C and $C \cup e$ to only refer to program variables that were used in random assignments.
- All our random assignments sample from independent random variables. Since we never compare integer and real random variables, it suffices to compute the probability that a system of linear comparisons over integers with integer coefficients hold and the probability that a system of linear comparisons over reals with rational coefficients hold. We will now show that these probabilities can be computed and are pseudo-rational functions.

We remark that we only need to consider systems of inequalities. If a comparison in C is $u_1 \neq u_2$ then we can consider the systems $C_1 = (C \setminus \{u_1 \neq u_2\}) \cup \{u_1 < u_2\}$ and

$C_2 = (C \setminus \{u_1 \neq u_2\}) \cup \{u_2 < u_1\}$, compute probabilities of C_1 and C_2 separately and add them up to compute the probability that C holds. Thus, without loss of generality we can assume that C consists of only linear inequalities.

Probability of system of linear inequalities over integers. Let $\overline{Z} = (Z_1, \ldots, Z_n)$ be a discrete random variable taking values in $\mathbb{Z}^n$. Consider a finite system of linear inequalities C with integer coefficients and with n unknowns $Z_1, \ldots, Z_n$. A solution of C is a tuple $\overline{b} = (b_1, \ldots, b_n) \in \mathbb{Z}^n$ such that all inequalities in C are satisfied when each $Z_j \in C$ is replaced by b_j. Let $sol(C) \subseteq \mathbb{Z}^n$ denote the set of all solutions of C. The probability that $\overline{Z}$ satisfies C is said to be the probability of the event $E = \{\overline{Z} = \overline{b} \mid \overline{b} \text{ is a solution of } C\}$. We denote this probability by $\mathsf{Prob}(\overline{Z} \models C)$. We have the following:

Lemma 6.6. Let C be a finite system of linear inequalities with integer coefficients and with n unknowns $Z_1, \ldots, Z_n$. Let $Z_1 = \mathsf{DLap}(a_1\epsilon, \mu_1)$, ..., $Z_n = \mathsf{DLap}(a_n\epsilon, \mu_n)$ be mutually independent discrete Laplacians such that for each $1 \leq j \leq n$, a_j is a strictly positive rational number and μ_j is an integer. Let $\overline{Z} = (Z_1, \ldots, Z_n)$. There is a pseudo-rational function $rt_{\overline{Z},C}$ in ϵ such that $\mathsf{Prob}(\overline{Z} \models C) = rt_{\overline{Z},C}$. The function $rt_{\overline{Z},C}$ can be computed from $C, (a_1, \mu_1), \ldots, (a_n, \mu_n)$.

Proof. For, each $1 \leq j \leq n$, consider $Y_j = \mathsf{DLap}(a_j\epsilon, 0)$. It is easy to see that Z_j has the same distribution as $Y_j + \mu_j$. Now consider the system of inequalities C' in which each Z_j is replaced by $Y_j + \mu_j$. Let $Y = (Y_1, \ldots, Y_n)$. It is easy to see that $\mathsf{Prob}(\overline{Z} \models C) = \mathsf{Prob}(\overline{Y} \models C')$. This observation implies that it suffices to prove the Lemma in the special case that each $\mu_j = 0$. Thus, for the rest of the proof we assume that the $\mu_j = 0$.

Now, consider a set $\mathsf{pos} \subseteq \{1, \ldots, n\}$. Let C_{pos} be the system of inequalities $C \cup \{Z_j \geq 0 \mid j \in \mathsf{pos}\} \cup \{Z_j < 0 \mid j \notin \mathsf{pos}\}$. It is easy to see that the set of solutions of C is the *disjoint* union $\cup_{\{\mathsf{pos} \subseteq 1, \ldots, n\}} C_{\mathsf{pos}}$. Thus, it suffices to the prove that for each $\mathsf{pos} \subseteq \{1, \ldots, n\}$, $\mathsf{Prob}(\overline{Z} \models C_{\mathsf{pos}})$ is a pseudo-rational function that can be computed.

Consider the system of inequalities C'_{pos} obtained from C_{pos} by replacing each Z_j by Y_j for $j \in \mathsf{pos}$ and by $-Y_j$ for $j \notin \mathsf{pos}$. Let $Y = (Y_1, \ldots, Y_n)$. From the fact that Laplacians are symmetric distributions, it follows each Y_j has the same distribution as Z_j. Thus, $\mathsf{Prob}(\overline{Z} \models C_{\mathsf{pos}}) = \mathsf{Prob}(\overline{Y} \models C'_{\mathsf{pos}})$. Observe that the set of solutions of C'_{pos} are a subset of $\mathbb{N}^n$. Without loss of generality, we can also assume that the terms in each inequality of C'_{pos} are rearranged so that the constant terms in C'_{pos} and the coefficients of the variables Y_j are natural numbers, ie, non-negative integers.

Therefore, C'_{pos} is a system of linear inequalities with natural number coefficients whose solutions are non-negative natural numbers. For such system of inequalities, the set of

solutions can be written as a *disjoint* union of *simple linear sets* [25]; a set $S \subseteq \mathbb{N}^n$ is said to be *linear* if there are tuples $\overline{b}_0, \overline{p}_1, \ldots, \overline{p}_m \in \mathbb{N}^n$ such that $S = \{\overline{b}_0 + \sum_{i=1}^{m} k_i \overline{p}_i |$ for each i, $k_i \in \mathbb{N}\}$ and *simple* each $\overline{b} \in S$ has a unique representation as a sum $\overline{b}_0 + \sum_{i=1}^{m} k_i \overline{p}_i$. $\overline{b}_0$ is said to be the offset of S and $\overline{p}_1, \ldots, \overline{p}_m$ the periods of S. From the fact that the set of solutions of C'_{pos} can be written as a *disjoint* union of *simple linear sets*, it follows that it suffices to show that $\mathsf{Prob}(\overline{Y} \in S \,|\, S$ is simple linear) is a pseudo-rational function in ϵ. In order to show this we need a couple of additional notations.

For two n-tuples $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{y} = (y_1, \ldots, y_n)$, $\overline{x} \cdot \overline{y}$ will denote the sum $\sum_{j=1}^{n} x_j y_j$. Secondly, we will denote the tuple $(a_1, \ldots, a_n)$ by $\overline{a}$.

Fix a simple semilinear set S. Let $\overline{b}_0$ be its offset and $\overline{p}_1, \ldots, \overline{p}_m$ its periods. From the fact that each $\overline{b} \in S$ has a unique representation as a sum $\overline{b}_0 + \sum_{i=1}^{m} k_i \overline{p}_i$, it follows that

$$\begin{array}{rcl} \mathsf{Prob}(\overline{Y} \in S) & = & \sum_{k_1 \in \mathbb{N}} \cdots \sum_{k_m \in \mathbb{N}} \mathsf{Prob}(\overline{Y} = \overline{b}_0 + \sum_{i=1}^{m} k_i \overline{p}_i) \\ & = & \sum_{k_1 \in \mathbb{N}} \cdots \sum_{k_m \in \mathbb{N}} e^{-\epsilon(\overline{b}_0 \cdot \overline{a} + k_1 \overline{p}_1 \cdot \overline{a} + \cdots + k_m \overline{p}_m \cdot \overline{a})} \\ & = & (e^{-\epsilon \overline{b}_0 \cdot \overline{a}})(\sum_{k_1 \in \mathbb{N}} e^{-\epsilon k_1 \overline{p}_1 \cdot \overline{a}}) \cdots (\sum_{k_m \in \mathbb{N}} e^{-\epsilon k_m \overline{p}_1 \cdot \overline{a}}) \\ & = & (e^{-\epsilon \overline{b}_0 \cdot \overline{a}})(\frac{1}{1 - e^{-\epsilon \overline{p}_1 \cdot \overline{a}}}) \cdots (\frac{1}{1 - e^{-\epsilon \overline{p}_m \cdot \overline{a}}}) \end{array}$$

The latter is clearly a pseudo-rational function in ϵ. *q.e.d*

Probability of system of linear inequalities over reals. Let $\overline{R} = (R_1, \ldots, R_n)$ be a continuous random variable taking values in $\mathbb{R}^n$. Consider a finite system of linear inequalities C with rational coefficients and with n unknowns $R_1, \ldots, R_n$. As in the case of discrete random variables , we can define $sol(C) \subseteq \mathbb{R}^n$, the set of solutions, and $\mathsf{Prob}(\overline{R} \models C)$, the probability that $\overline{R}$ satisfies C. We have the following result.

Lemma 6.7. Let C be a finite system of linear inequalities with rational coefficients and with n unknowns $R_1, \ldots, R_n$. Let $R_1 = \mathsf{Lap}(a_1\epsilon, \mu_1)$, ..., $R_n = \mathsf{Lap}(a_n\epsilon, \mu_n)$ be mutually independent Laplacians such that for each $1 \leq j \leq n$, a_j is a strictly positive rational number and μ_j is a rational number. Let $\overline{R} = (R_1, \ldots, R_n)$. There is a pseudo-rational function $rt_{\overline{R},C}$ in ϵ such that $\mathsf{Prob}(\overline{R} \models C) = rt_{\overline{R},C}$. The function $rt_{\overline{R},C}$ can be computed from $C, (a_1, \mu_1), \ldots, (a_n, \mu_n)$.

Proof. As in the proof of Lemma 6.6, it suffices to consider the case when each $\mu_i = 0$ and to show that the probability measure of the set $Sol = sol(C) \cap \{(b_1, \ldots, b_n) \mid b_i \in \mathbb{R}^{>0}\}$ is a computable pseudo-rational function.

Since $\overline{R}$ is continuous, we can also assume that each inequality is of the form $\leq$. There are computable finite sets $S_1, \ldots, S_m$ such that (See [26])

1. $Sol = S_1 \cup \ldots S_m$,
2. $S_i \neq S_j$ for $i \neq j$, and
3. Each S_i is a positive repetitive polyhedra. $S \subseteq (\mathbb{R}^{>0})^n$ is said to be a positive repetitive polyhedra if there are constants h_0^-, h_0^+ and functions $h_1^-(x_1)$, $h_1^+(x_1)$, $h_2^-(x_1, x_2)$, $h_2^+(x_1, x_2)$, $\ldots$, $h_{n-1}^-(x_1, x_2, \ldots x_{n-1})$, $h_{n-1}^+(x_1, x_2, \ldots x_{n-1})$ such that
   - $S_i = \begin{array}{l}\{(x_1, \ldots, x_n) \mid h_0^- \leq x_1 \leq h_0^+, \ldots, \\ h_{n-1}^-(x_1, \ldots x_{n-1}) \leq x_n \leq h_{n-1}^+(x_1, \ldots x_{n-1})\}.\end{array}$
   - h_0^- is a rational number > 0.
   - h_0^+ is either ∞ or a rational number.
   - For each $1 \leq j \leq n$, h_j^- is a linear function in its arguments. h_j^- has rational coefficients.
   - For each $1 \leq j \leq n$, h_j^+ is either ∞ or a linear function in its arguments. h_j^+ has rational coefficients in the latter case.

Thanks to conditions (1) and (2) above, it suffices to show that for any positive repetitive polyhedra S, the probability measure of the event $\{\overline{R} = \overline{b} \mid \overline{b} \in S\}$ is a pseudo-rational function.

Fix S and let $h_0^-, h_0^+, h_1^-, h_1^+, \ldots, h_{n-1}^-, h_{n-1}^+$ be as above, The measure of the event $\{\overline{R} = \overline{b} \mid \overline{b} \in S\}$ can be computed using the nested integral

$$F = \int_{h_0^-}^{h_0^+} f_{a_1}(x_1) \int_{h_1^-}^{h_1^+} f_{a_2}(x_2) \cdots \int_{h_{n-1}^-}^{h_{n-1}^+} f_{a_n}(x_n)\, dx_n \cdots dx_1$$

where $f_{a_i} = \frac{a_i \epsilon}{2} e^{-a_i \epsilon}$ is the pdf of R_i and the arguments of h_i^+, h_i^- are omitted for readability.

For $1 \leq j \leq n$, let I_j be the nested integral

$$I_j = \int_{h_{j-1}^-}^{h_{j-1}^+} f_{a_j}(x_j) \cdots \int_{h_{n-1}^-}^{h_{n-1}^+} f_{a_n}(x_n)\, dx_n \cdots dx_j.$$

We claim by induction on $k = n - j$ that I_j is a finite sum of terms of the form

$$a \epsilon^m e^{b\epsilon} (x_1^{m_1} e^{\epsilon b_1 x_1}) \ldots (x_{j-1}^{m_{j-1}} e^{\epsilon b_{j-1} x_{j-1}})$$

where a, b, b_i are rational numbers (including negative numbers) m is an integer, and m_i are natural numbers. We will assume that the sum is always presented in simplest form, namely, that all cancellations have already taken place in the sum.

Clearly the claim is true when $k = 0$. Suppose that the claim is true for $k = k_0$. Let $j_0 = n - k_0$. Suppose

$$w = a\epsilon^m e^{b\epsilon}(x_1^{m_1} e^{\epsilon b_1 x_1}) \ldots (x_{j_0-1}^{m_{j_0-1}} e^{\epsilon b_{j_0-1} x_{j_0-1}})$$

is a summand in I_{j_0}. Let $k = k_0 + 1$ and $j = n - k = n - k_0 - 1 = j_0 - 1$.

Consider the indefinite integral

$$\begin{aligned} J &= \int f_{a_{j_0-1}} w \, dx_{j_0-1} \\ &= \int \tfrac{a_{j_0-1}\epsilon}{2} e^{-a_{j_0-1}\epsilon} w \, dx_{j_0-1} \\ &= \tfrac{a a_{j_0-1}}{2} \epsilon^{m+1} e^{b\epsilon} (x_1^{m_1} e^{\epsilon b_1 x_1}) \ldots (x_{j_0-2}^{m_{j_0-2}} e^{\epsilon b_{j_0-2} x_{j_0-2}}) \\ &\qquad \int x_{j_0-1}^{m_{j_0-1}} e^{\epsilon(b_{j_0-1} - a_{j_0-1}) x_{j_0-1}} dx_{j_0-1} \end{aligned}$$

Let

$$J' = \int x_{j_0-1}^{m_{j_0-1}} e^{\epsilon(b_{j_0-1} - a_{j_0-1}) x_{j_0-1}} dx_{j_0-1}.$$

Now, if $b_{j_0-1} - a_{j_0-1} = 0$ then

$$J' = \frac{x_{j_0-1}^{m_{j_0-1}+1}}{m_{j_0-1+1}}.$$

If $b_{j_0-1} - a_{j_0-1} \neq 0$ then by doing a change of variables $t = (b_{j_0-1} - a_{j_0-1})\epsilon x$, it is not too hard to show that

$$J' = \sum_{k=0}^{m_{j_0-1}} c_k \epsilon^{t_k} x^k e^{\epsilon(b_{j_0-1} - a_{j_0-1}) x_{j_0-1}}$$

where c_k is a rational number and t_k an integer for each k.

Thus, the indefinite integeral J is a sum, each of whose terms is of the form

$$a'\epsilon^{m'} e^{b'\epsilon} (x_1^{m'_1} e^{\epsilon b'_1 x_1}) \ldots (x_{j_0-1}^{m'_{j_0-1}} e^{\epsilon b'_{j_0-1} x_{j_0-1}}).$$

If $h^-_{j_0-2}$ and $h^+_{j_0-2}$ are linear functions, we get immediately that $I_j = \int_{h^-_{j_0-2}}^{h^+_{j_0-2}} f_{a_{j_0-1}} w \, dx_{j_0-1}$ is of the right form. The induction step follows in this case.

If $h^+_{j_0-2} = \infty$, and each b_j in a summand of J is strictly negative, then it is also easy to see that the induction step follows. Apriori, it seems that there might be a problem when $b_j \geq 0$ as in this case, I_j will evaluate to either ∞ or $-\infty$. This, however, will contradict the fact that the nested integral F defines probability of an event (and hence is bounded above by 1). Thus, if $h^+_{j_0-2} = \infty$ then b_j must be strictly negative.

The claim immediately implies that the $Sol = sol(C) \cap \{(b_1, \ldots, b_n) \mid b_i \in \mathbb{R}^{>0}\}$ is a pseudo-rational function. *q.e.d*

q.e.d

We conclude with the main result of our report.

Theorem 6.8. The Fixed Parameter Differential Privacy and Differential Privacy problems are decidable for DiPWhile programs P_ϵ and definable functions δ.

Proof. Let in and out be arbitrary valuations to input and output variables, respectively. Observe that the function $\epsilon \mapsto \mathsf{Prob}(P_\epsilon(\mathsf{in}) = \mathsf{out})$ is nothing but $\mathsf{Prob}(z_0, Z')$ in $[\![P_\epsilon]\!]$, where z_0 is the initial state corresponding to valuation in, and Z' is the set of all terminating states that have valuation out for output variables. Decidability then follows from Theorems 6.3, 6.2, and 4.4. *q.e.d*

CHAPTER 7: DiPC TOOL

We have implemented the decidability algorithm for programs in a tool, named DiPC. In this chapter, we shall present the implementation details of the tool. The tool takes as input an acyclic (i.e. no while loops) query program with finite domain variables and real variables (i.e. no integer variables) and a list of adjacent input pairs and checks for $(k\epsilon, \delta(\epsilon))$-Differential Privacy for a given k and all $\epsilon > 0$. Additionally, the tool can also check Differential Privacy for ϵ belonging to some set $S \subseteq \mathbb{R}^{>0}$ or a fixed ϵ value.

Given a program and an adjacency relation, DiPC outputs true if the program is differentially private and outputs a counterexample if it is not. The tool works in two phases. In the first phase, the tool parses the program, computes symbolic expressions that capture the output distribution for every input, and identifies inequalities that must hold for differential privacy. The symbolic expressions for the probability computation, and the logical constraints that must hold, are written in a Wolfram Mathematica® script. In the second phase, Mathematica® is run to perform the symbolic computations and check the results.

In Section 7.1, we shall present various approaches taken during tool development to optimize the runtime of the tool. Currently, the tool uses Approach 3 (Sec. 7.1.3) for ϵ-Differential Privacy and allows the user to choose between Approach 4 (Sec. 7.1.4) and Approach 5 (Sec. 7.1.5) for (ϵ, δ)-Differential Privacy.

7.1 STAGES IN TOOL DEVELOPMENT

The input query program for DiPC takes finite domain variables as input and outputs (possibly multiple) finite domain variables as output. The program can further use real variables as local variables. The real variables and finite domain variables can be assigned values based on probabilistic distribution (laplace and exponential distributions).

As discussed above, the Mathematica® script generated contains two major sets of operations: the first operation set, where the output probability distribution for each input is computed and a second set of commands, where the probabilities of outputs are compared for each input adjacency pair. While the algorithm for generating the second set of commands remained the same over the course of the work, multiple alternative algorithms were tried for the output probability generation, in attempts to improve the runtime of the tool.

To compute the probability distribution of the output, DiPC adopts a top-down approach to start from the input and run down the program to generate the output. Due to the existence of branching statements in the program, the output can be modelled as a tree,

with new branches generated at every branching statement.

In our algorithm, we have branching in two ways: the if-else statements, where the comparisons may involve finite or infinite domain variables and the probabilistic assignment to integer variables (`TDLap`). To deal with the latter form of branching, we consider each possible integer assignment and assign the corresponding probability to the output generated by the branch. In this fashion, all variables, except the real variables are always assigned a unique value in every branch. Hence, If-Else comparisons not involving real variables can be deterministically decided using the variable values.

The only remaining obstacle is handling comparisons over real variables.

7.1.1 Approach 1: Total Order over Real Variables and Probability[] Command

Given the real variables in the program, we can generate the output distribution for each total order over the real variables. Comparisons over real variables can be resolved based on the total order being considered currently.

Thus, for a given total order, we can now compute the output distribution for the input. Mathematica® allows us to compute the probability of a total order given the probability distributions for the real variables using the `Probability[]` command. We can now iterate over all possible total orders and compute the overall output distribution for the input.

This approach worked well for small number of queries. But it was found that the `Probability[]` command was quite slow and would take hours for probability computation with four real variables.

7.1.2 Approach 2: Partial Order over Real Variables and Probability[] Command

A straightforward improvement to the previous approach would be to only iterate over the partial orders required, instead of fixing order over all the real variables. This optimization would greatly reduce the number of `Probability[]` commands, since we would now have only one `Probability[]` call for each partial order, while previously for each total order corresponding to the partial order, we would have one call.

Partial order for the program was modelled as a DAG over the variables and every time a comparison over real variables is performed in the program execution, if the variables are unordered so far, we add both the possible orders between the variables and explore the resulting two branches, and otherwise, we explore the corresponding branch.

The `Probability[]` command was used to compute the probability of the partial order generated. It turned out that this method too, was pretty slow, with the `Probability[]`

command taking hours for 3-4 variables.

7.1.3 Approach 3: Partial Order over Real Variables and Integrate[] Command

Owing to the slow runtime of the `Probability[]` command, we chose to use integration to compute the probability of the real variable ordering.

Similar to the previous approach, we compute the partial order alongside the output computation. Given the partial order, we then generate all possible total orders which satisfy the partial order. If we can compute the probability of each such total order, the probability of the partial order would be the sum of probabilities of the total orders.

Let one such total order generated be $x_1 < x_2 < x_3 < x_4 < x_5$. To compute the probability of this order, we first compute $P(x_5 > x) = \int_x^\infty f_5(y)dy$, where $f_5(.)$ is the probability density function (p.d.f) for x_5. Then, we can compute $P(x_5 > x_4 > x) = \int_x^\infty P(x_5 > y)f_4(y)dy$ (NOTE: The equality follows because the integrand is the probability that x_4 is y and x_5 is more than y). Iterating in a similar fashion, we can further compute $P(x_5 > x_4 > x_3 > x)$ $\cdots P(x_5 > x_4 > x_3 > x_2 > x_1 > x)$. Now, we have, $P(x_5 > x_4 > x_3 > x_2 > x_1) = Lt_{x\to-\infty}P(x_5 > x_4 > x_3 > x_2 > x_1 > x)$.

Note that we also need to deal with the case where the real variables are not assigned probabilistically, but are given a constant value. This can be handled by adjusting the upper and lower limits of the neighbouring integrations. For example, $P(x_2 > 1 > x_1)$ can be computed as $\Big(Lt_{x\to1}P(x_2 > x)\Big).\Big(Lt_{x\to-\infty}\int_x^1 f_1(y)dy\Big)$

For further optimization, before computing the set of total orders which satisfy the given partial order, we can separate out the connected components in the partial order and compute total orders for each component separately, to further reduce the depth of integration.

This is the algorithm implemented in the current version of the tool for ϵ-Differential Privacy.

7.1.4 Approach 4: (ϵ, δ)-Differential Privacy using Subsets of Outputs

The definition of Differential Privacy requires that the probability of every subset of outputs given two adjacent inputs must be similar. However, in the case of ϵ-Differential Privacy, it suffices to check the condition for individual outputs, which is what was done in the previous approaches.

However, while adding the ability of (ϵ, δ)-Differential Privacy, we included the ability to compare over subsets of outputs. The probability of a subset of outputs is simply computed as sum of the individual output probabilities.

It turned out that Mathematica® takes a significantly longer time for (ϵ, δ)-Differential Privacy checks with subsets of outputs.

7.1.5 Approach 5: (ϵ, δ)-Differential Privacy using Optimal Subset Construction

An alternate approach was to construct a subset which was more likely to exceed the error bound. To achieve this, when the output probabilities of adjacent inputs are computed, we compute the quantity $\sum_{o_i \in Outputs} max(P_\epsilon(o_i|inp_1) - e^\epsilon.P_\epsilon(o_i|inp_2), 0)$. To elaborate, for each output, we look at the probability difference over the two inputs (which is a function over epsilon) and take the maximum of this difference and zero (which is another function over epsilon). This difference is added up over all outputs to give us another function over epsilon, which gives the maximum possible error over any output subset as a function of epsilon.

We then check if this function exceeds δ for any value of epsilon. While this approach is faster than the previous approach, the tool can currently only generate the input pair violating the inequality, but can not produce the output subset. The previous approach, though slower, can generate both the input pair and the output subset corresponding to the privacy violation.

Currently, the tool supports both Approach 4 and Approach 5 as algorithms for (ϵ, δ)-Differential Privacy.

7.2 DIPC USAGE

In this section, we shall briefly discuss the syntax and usage of DiPC. DiPC takes as input two files. The first input file, named `input.cpp`, declares a function `compare` which defines the adjacency relation between the possible inputs. `compare` is a boolean function, which given a pair of inputs as arrays, returns true if the pair are adjacent and false if otherwise.

The second input file, given as a commandline input, contains the input program to be used. Before diving into the syntax of the input program, a few important notes about the tool implementation are given below:

- The tool uses a rudimentary compiler which requires that different tokens in the program be separated by whitespace, i.e., `"var=var1+var2;"` needs to be written as `"var = var1 + var2;"`.
- Real variables in the program are treated as constant variables. Each real variable is assumed to be assigned only once in each branch of the program.

- The tool further assumes that the input program is acyclic (i.e. there are no while loops in the program).
- It is assumed that every branch of the program terminates with a return statement, and all the return statements in the program return the same number of variables and the data types of variables in each position is preserved across all return statements.

We shall now discuss the format of the second input file. The first line must contain two integers separated by a whitespace. These two integers are respectively, the lower and upper bound of the integer variables for the program. The rest of the file would be the input program written according to the syntax discussed in the following subsection.

7.2.1 DiPC Input Program Syntax

1. The first lines denote the input to the program. Each of the input variables are listed in a separate line. The syntax is:

   ```
   input <var_type> <var_name> <min_val> <max_val>;
   ```

   or

   ```
   input <var_type> <var_name>;
   ```

   - `<var_type>` can be "int" or "bool".
   - `<var_name>` is the name of the variable, which can be any string with no newline or space characters and the first character is an alphabet.
   - `<min_val>` to `<max_val>` is the range we allow the inputs to take for this variable.

   Eg: `input int a 5 7;`

2. The remaining program allows the following commands:

   - `<var_type> <var_name>;` This is the variable declaration statement.
     - `<var_type>`, the type of the variable, can be "int", "bool" or "real", denoting integer, boolean and real variables respectively.
     - `<var_name>`, the name of the variable, can be any string with no newline or space characters and the first character is an alphabet.

     Eg: `int val;`
   - `<var_name> = <constant>;` This is the constant assignment statement. The allowed values depend on the type of the variable:

  - int: Any value in the RANGE defined in the program.
  - bool: true or false.
  - real: Any real value.

  Eg: `val = -5;`

- `<var_name1> = <var_name2>;` This is the variable assignment statement. The types of both the variables must be the same.

  Eg: `val1 = val2;`

- `<real_var> = <int_var>;` Using this statement, we can assign the value of the integer variable to the real variable.

  Eg: `rl = val;`

- `<var_name> = <var_name1> <op> <var_name2>;` This is the arithmetic operation command.
  - The variable `<var_name>` must be an integer variable or a boolean variable.
  - We allow `+, -, *, /, %, <, <=, >=, >` and `==` between integer variables, and `&&` and || between boolean variables.

  Eg: `val = val1 + val2;`

- `<bool_var1> = ! <bool_var2>` This is the negation statement.

  Eg: `bl1 = ! bl2;`

- `<real_var> = Lap "<lin_in_eps" <int_var>;` This is the Laplacian assignment statement.
  - `<lin_in_eps>` is a linear expression in eps and other variables defined in the program, but every token must be space separated, i.e., "2*eps+3" needs to be written as "2 * eps + 3".
  - `<int_var>` This is the variable holding the value of the mean of the Laplacian distribution.

  Eg: `rl = Lap "val * eps + 3" val;`

- `return <var1> <var2> ... <vark>;` This is the return statement. We can return an list of variables. But the length of the array and the type of each variable must be constant across all return statements in the program. `<vari>` can be an integer variable or a boolean variable.

  Eg: `return val1 val2 val3;`

- `if <bool_var> then <program1> else <program2> fi;` This is the if-else statement.
  - `<bool_var>` must be a boolean variable.
  - `<program1>` and `<program2>` are lists of commands.

  Eg:
  ```
  if b then val = val + a;
  b = b + a;
  return val;
  else val = val + b;
  a = a + b;
  fi;
  ```
- `if <bool_var> then <program1> fi;` This is the if statement.
  - `<bool_var>` must be a boolean variable.
  - `<program1>` is a list of commands.

  Eg:
  ```
  if b then val = val + a;
  b = b + a;
  return val;
  fi;
  ```
- `<int_var> = TDLap "<lin_in_eps>" <var1> <var2> <var3>;` This is the truncated discrete laplacian statement.
  - `<var1>` is the variable storing value of the variance. `<var2>` is the variable storing lower cutoff for the distribution. `<var3>` is the variable storing upper cutoff for the distribution. `<lin_in_eps>` is a linear expression in eps, as in the Laplacian statement.

  Eg: `val = TDLap "val * eps + 3" val1 val2 val3;`
- `Prob{<math_exp>} <statement>` This is the probabilistic statement.
  - `<math_exp>` is a Mathematica® Expression for the probability with which this statement must be executed. This probability may depend on ϵ
  - `<statement>` is the statement to be executed, which may be one of the previous types of statements

  The command `<statement>` is executed with probability `<math_exp>` and not executed with probability `1-<math_exp>`.

  Eg: `Prob{1/(Exp[eps]+1)} out1 = ! q1;`

3. NOTE: We don't have assignments of the form `var1 = var2 + 3`. In the arithmetic statement, both the operands must be variables. `"var1 = var2 + 3"` must be written as `"int three; three = 3; var1 = var2 + three;"`.

7.2.2 DiPC Commandline Parameters

We shall briefly discuss the different kinds of Differential Privacy checks the DiPC tool can perform. By default, given an input program file, the tool performs ϵ-Differential Privacy check of the program, for all $\epsilon > 0$. The following are the additional commandline parameters, which can be used for performing other Differential Privacy checks.

- `FRAC`: Given a value `FRAC = k`, the tool would perform $k\epsilon$-Differential Privacy check.
- `EPS`: Given a value `EPS` = ϵ_0, the tool would perform ϵ-Differential Privacy only for $\epsilon = \epsilon_0$.
- `RANGE`: Given `RANGE = range`, where `range` is a Mathematica® expression of a variable range for ϵ, the tool would perform ϵ-Differential Privacy over $\epsilon \in$ `range`.
- `DELTA`: Given `DELTA` = δ, where δ is a function of ϵ, the tool would perform (ϵ, δ)-Differential Privacy check.
- `APPROACH`: This is an option for (ϵ, δ)-Differential Privacy check. `APPROACH` can be assigned 0 or 1. If `APPROACH = 0`, then the tool uses algorithm in Approach 4 (Sec. 7.1.4) and otherwise, uses the algorithm in Approach 5 (Sec. 7.1.5).

CHAPTER 8: EXPERIMENTS

We used various examples to measure the effectiveness of our tool. These include SVT [20, 27], Noisy Maximum [11], Noisy Histogram [11] and Randomized Response [19]. Pseudocodes for all variants of these examples are discussed in Section 8.1. Though the pseudo-codes don't strictly adhere to the syntax of DiPWhile programs, they can easily be rewritten to fit the syntax.

8.1 EXAMPLES

Sparse Vector Technique (SVT) We looked at six different variants of the Sparse Vector Technique (SVT). Algorithms addressed as SVT1-6, are Algorithms 1-6 in [20], respectively. SVT1 and SVT3 were previously introduced in this report as Algorithm 8.1 and 8.2 and Algorithm 3.2, respectively. The adjacency relation Φ we used is given by $((t_1, q_1), (t_2, q_2)) \in \Phi$ if and only $t_1 = t_2$ and $|q_1[i] - q_2[i]| \leq 1$ for all i. While SVT1 and SVT2 are differentially private, the other four variants are not. We will present counterexamples for all four of these variants in Section 8.2.

The pseudocode for the six variants of SVT are given in Figures 8.1 and 8.2. In these programs, the array Q represents the input queries. The array a represents the output array, $\perp$ represents False and $\top$ represents True. Note that $\mathcal{U}$ can be identified with tuples $(t, \overline{a})$ where t is an integer and $\overline{a}$ is vector of integers; $t, \overline{a}$ are the values of T and Q respectively.

Noisy Maximum Noisy maximum algorithms are a differentially private way to compute different statistical measures for a given set of queries. Algorithms addressed as NMax1-4 are Algorithms 5-8, respectively, in [11]. Algorithms NMax1 and NMax2 are mechanisms to compute the index of the query with maximum value after adding a Laplacian (or exponential) noise. Inputs Q_1 and Q_2 are considered adjacent iff $|Q_1[i] - Q_2[i]| \leq 1$ for all i. Under this relation, Algorithms NMax1 and NMax2 are both ϵ-differentially private. Algorithms NMax3 and NMax4 are variants to print the maximum value instead of the index. These variants are shown to be not differentially private in Section 8.2.

Histogram Algorithms Histogram algorithms also target computing statistical measures on queries in a differentially private manner. Algorithms referred to as Hist1-2 here are Algorithms 9-10 in [11]. Algorithm Hist1 and Hist2 are variants of noisy maximum, where we return the histogram, instead of the maximum. Under the above adjacency relation where

(SVT1) First Instantiation of SVT

```
Input: q[1 : N]
Output: out[1 : N]
r_T ← Lap(ε/2Δ, T)
count ← 0
for i ← 1 to N do
    r ← Lap(ε/4cΔ, q[i])
    b ← r ≥ r_T
    if b then
        out[i] ← ⊤
        count ← count + 1
        if count ≥ c then
            exit
        end
    else
        out[i] ← ⊥
    end
end
```

(SVT2) Second Instantiation of SVT

```
Input: q[1 : N]
Output: out[1 : N]
r_T ← Lap(ε/2cΔ, T)
count ← 0
for i ← 1 to N do
    r ← Lap(ε/4cΔ, q[i])
    b ← r ≥ r_T
    if b then
        out[i] ← ⊤, r_T ← Lap(ε/2cΔ, T)
        count ← count + 1
        if count ≥ c then
            exit
        end
    else
        out[i] ← ⊥
    end
end
```

Figure 8.1: Sparse Vector Technique Algorithms

Q_1 and Q_2 are adjacent if $|Q_1[i]-Q_2[i]| \leq 1$ for all i, both these variants are not ϵ-differentially private. However, if we consider an alternative definition for the adjacency relation, where Q_1 and Q_2 are adjacent iff $\sum_i \Big(|Q_1[i] - Q_2[i]|\Big) \leq 1$, then Hist1 is ϵ-differentially private but Hist2 still is not. All experiments listed in Section 8.2 for Algorithms NMax1 and NMax2 were run using the second adjacency relation.

Pseudocode for the Noisy Maximum and Histogram variants has been given in Figures 8.3 and 8.4. The variable Q is the set of input queries.

Randomized Response All the previous algorithms use the Laplace mechanism. Randomized Response [19], on the other hand, uses discrete probabilities. In this algorithm, given a set of Boolean input queries, we flip each input query with a probability of $\frac{1}{e^\epsilon+1}$ and output the resulting outcome. We also consider a non-private version where the input query is flipped with probability $\frac{1-\epsilon}{2}$. We will refer to these algorithms as Rand1 and Rand2 henceforth.

The pseudocode for the Randomized Response algorithm [19] can be found in Figure 8.5. Each boolean input query is flipped with a probability of $\frac{1-\epsilon}{2}$ and the resultant query set is returned as output. We consider two versions. In the second version, we change the parameter of the Bernoulli distribution and obtain a non-private algorithm.

(SVT3) Third Instantiation of SVT

```
Input: q[1 : N]
Output: out[1 : N]
r_T ← Lap(ε/2Δ, T)
count ← 0
for i ← 1 to N do
    r ← Lap(ε/2cΔ, q[i])
    b ← r ≥ r_T
    if b then
        out[i] ← Disc_seq(r)
        count ← count + 1
        if count ≥ c then
            exit
        end
    else
        out[i] ← ⊥
    end
end
```

(SVT4) Fourth Instantiation of SVT

```
Input: q[1 : N]
Output: out[1 : N]
r_T ← Lap(ε/4Δ, T)
count ← 0
for i ← 1 to N do
    r ← Lap(3ε/4Δ, q[i])
    b ← r ≥ r_T
    if b then
        out[i] ← ⊤
        count ← count + 1
        if count ≥ c then
            exit
        end
    else
        out[i] ← ⊥
    end
end
```

(SVT5) Fifth Instantiation of SVT

```
Input: q[1 : N]
Output: out[1 : N]
r_T ← Lap(ε/2Δ, T)
for i ← 1 to N do
    r ← q[i]
    b ← r ≥ r_T
    if b then
        out[i] ← ⊤
    else
        out[i] ← ⊥
    end
end
```

(SVT6) Sixth Instantiation of SVT

```
Input: q[1 : N]
Output: out[1 : N]
r_T ← Lap(ε/2Δ, T)
for i ← 1 to N do
    r ← Lap(ε/2Δ, q[i])
    b ← r ≥ r_T
    if b then
        out[i] ← ⊤
    else
        out[i] ← ⊥
    end
end
```

Figure 8.2: Sparse Vector Technique Algorithms

Sparse Sparse is a variant of SVT that is discussed in [27]. Pseudocode for this algorithm, referred to henceforth as Sparse, can be found as Figure 8.6. Our reason for considering this example is to demonstrate our tool's ability to handle (ϵ, δ)-differential privacy (see Section 8.2).

(NMax1) Correct Noisy Max with Laplacian Noise

Input: $q[1:N]$
Output: *out*

NoisyVector $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  NoisyVector[i] $\leftarrow \mathsf{Lap}(\frac{\epsilon}{2}, q[i])$
end
out $\leftarrow$ argmax(NoisyVector)

(NMax2) Correct Noisy Max with Exponential Noise

Input: $q[1:N]$
Output: *out*

NoisyVector $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  NoisyVector[i] $\leftarrow \mathsf{Lap}^{+}(\frac{\epsilon}{2}, q[i])$
end
out $\leftarrow$ argmax(NoisyVector)

(NMax3) Incorrect Noisy Max with Laplacian Noise

Input: $q[1:N]$
Output: *out*

NoisyVector $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  NoisyVector[i] $\leftarrow \mathsf{Lap}(\frac{\epsilon}{2}, q[i])$
end
out $\leftarrow \mathsf{Disc}_{\mathsf{seq}}$(max(NoisyVector))

(NMax4) Incorrect Noisy Max with Laplacian Noise

Input: $q[1:N]$
Output: *out*

NoisyVector $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  NoisyVector[i] $\leftarrow \mathsf{Lap}^{+}(\frac{\epsilon}{2}, q[i])$
end
out $\leftarrow \mathsf{Disc}_{\mathsf{seq}}$(max(NoisyVector))

Figure 8.3: Noisy Max Algorithms

(Hist1) Noisy Histogram

Input: $q[1:N]$
Output: $out[1:N]$

NoisyVector $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  NoisyVector[i] $\leftarrow \mathsf{Lap}(\epsilon, q[i])$
end
out $\leftarrow \mathsf{Disc}_{\mathsf{seq}}$(NoisyVector)

(Hist2) Noisy Histogram, Wrong Scale

Input: $q[1:N]$
Output: $out[1:N]$

NoisyVector $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  NoisyVector[i] $\leftarrow \mathsf{Lap}(\frac{1}{\epsilon}, q[i])$
end
out $\leftarrow \mathsf{Disc}_{\mathsf{seq}}$(NoisyVector)

Figure 8.4: Noisy Histogram Algorithms

8.2 EXPERIMENTAL RESULTS

We ran all the experiments on an octa-core Intel® Core i7-8550U @ 1.8gHz CPU with 8GB memory. The tool is implemented in C++ and uses Wolfram Mathematica®. As mentioned in Chapter 7, the tool works in two phases — in the first phase, a Mathematica® script is

(Rand1) Randomized Response

Input: $q[1:N]$
Output: $out[1:N]$

out $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  out[i] $\leftarrow \begin{cases} q[i] & probability = \frac{e^\epsilon}{1+e^\epsilon} \\ \neg q[i] & probability = \frac{1}{1+e^\epsilon} \end{cases}$
end

(Rand2) Non-private Randomized Response

Input: $q[1:N]$
Output: $out[1:N]$

out $\leftarrow$ []
for $i \leftarrow 1$ **to** N **do**
  out[i] $\leftarrow \begin{cases} q[i] & probability = \frac{1+\epsilon}{2} \\ \neg q[i] & probability = \frac{1-\epsilon}{2} \end{cases}$
end

Figure 8.5: Randomized Response Algorithm

(Sparse) SVT for $(\frac{\epsilon}{2}, \delta)$-Differential Privacy

Input: $q[1:N]$
Output: $out[1:N]$
$\sigma \leftarrow \frac{\epsilon}{2\sqrt{32c \ln \frac{1}{\delta}}}$

$\mathsf{r}_T \leftarrow \mathsf{Lap}(\sigma, T)$
$count \leftarrow 0$
for $i \leftarrow 1$ **to** N **do**
  $\mathsf{r} \leftarrow \mathsf{Lap}(\frac{\sigma}{2}, q[i])$
  $\mathsf{b} \leftarrow \mathsf{r} \geq \mathsf{r}_T$
  if b **then**
    $out[i] \leftarrow \top$, $\mathsf{r}_T \leftarrow \mathsf{Lap}(\sigma, T)$
    $count \leftarrow count + 1$
    if $count \geq c$ **then**
      exit
    end
  else
    $out[i] \leftarrow \perp$
  end
end

Figure 8.6: SVT Algorithm for $(\frac{\epsilon}{2}, \delta)$-Differential Privacy

produced with commands for all the output probability computations and the subsequent inequality checks and in the second phase, the generated script is run on Mathematica®. In all the following tables, we refer the times of the Script Generation Phase (i.e. Phase 1) as T1 and that of the Script Validation Phase (i.e. Phase 2) as T2.

Unless stated otherwise, all the experiments were run with the parameters $c = 1$, $\Delta = 1$

and discretization parameter $seq = (-1 < 0 < 1)$ wherever applicable. The range of input query values was $Dom = \{-1, 0, 1\}$ in all the experiments. The running times in all experiments were averaged over 3 runs of the tool.

As stated before, our tool can be used to both prove and disprove Differential Privacy. We ran our tool on all the algorithms discussed in Section 8.1. Table 8.1 shows the runtime of our tool for all the listed algorithms with 3 queries, with the result of the execution. We chose to use 3 queries because counterexamples for most of the programs which were not differentially private could be found with 3 queries: the only exception being SVT3. Majority of the time is taken for running the Mathematica® code. We also observed that most of the time spent by Mathematica® was in computing the output probability: the time to perform the inequality checks for adjacent inputs was relatively smaller. Consequently, programs which do not use real variables are much faster to run. Results in the table also show that the time taken for disproving Differential Privacy is lower than the time for proving Differential Privacy on average (SVT3 vs SVT4 or Hist1 vs Hist2). This is because the tool terminates on finding a counterexample. On the other hand, to prove differential privacy the tool has to check all inequalities.

Algorithm	Runtime (T1/T2)	ϵ-Diff. Private
SVT1	0s/825s	✓
SVT2	0s/768s	✓
SVT3	0s/3816s	✓
SVT4	0s/269s	✗
SVT5	0s/2s	✗
SVT6	0s/661s	✗
NMax1	0s/197s	✓
NMax2	0s/59s	✓
NMax3	0s/310s	✗
NMax4	1s/58s	✗
Hist1	0s/1450s	✓
Hist2	0s/55s	✗
Rand2	0s/0s	✗

Table 8.1: Runtime for 3 Queries for each algorithm searching over adjacency pairs and all $\epsilon > 0$, with parameters being [c=1, Δ=1, Domain={-1,0,1}, seq=(-1<0<1)]

Additionally, our tool also has the capability to produce counterexamples for the algorithms which are not Differentially Private. It is interesting to note that DiPC has the ability to compute the smallest counterexamples for algorithms, since it can both prove and disprove privacy definitively, unlike the statistical algorithms. Table 8.2 lists the smallest

counterexample for each non differentially private algorithm. Given a program and an adjacency relation, the tool automatically finds an ϵ, the pair of adjacent inputs, and the output value that demonstrate the violation of differential privacy. All four columns in the table were output by the tool. Further, we observe that the counterexamples found were much smaller, in number of queries, compared to those found in [11]. For example, algorithms NMax3 and NMax4 counterexamples need just 3 and 1 queries respectively, compared to the 5 queries required in [11]. Similarly, algorithm SVT5 has a counterexample with just 2 queries, as compared to the 10 queries.

Algo	|Q|	Output	Input-1	Input-2	ϵ	Runtime (T1/T2)
SVT3	5	[$\perp$ $\perp$ $\perp$ $\perp$ 0], seq=(0<1)	[-1 -1 -1 -1 -1]	[0 0 0 0 0]	27	18s/5042s
SVT4	2	[$\perp$ $\top$]	[-1 0]	[0 -1]	27/50	0s/81s
SVT5	2	[$\perp$ $\top$]	[-1 0]	[-1 -1]	27	0s/2s
SVT6	3	[$\perp$ $\perp$ $\top$]	[-1 -1 0]	[0 0 -1]	67/92	0s/661s
NMax3	3	-1, seq=(-1<0<1)	[-1 -1 -1]	[0 0 0]	27	0s/310s
NMax4	1	0, seq=(-1<0<1)	[-1]	[0]	27	0s/2s
Hist2	1	[-1], seq=(-1<0<1)	[-1]	[0]	9/34	0s/3s
Rand2	1	[$\perp$]	[$\perp$]	[$\top$]	9/34	0s/0s

Table 8.2: Smallest Counterexample found for each non-differentially private algorithm, searching over all adj. pairs and $\epsilon > 0$, with parameters being [c=1, Δ=1, Domain={-1,0,1}]

The scalability of the tool in runtime is an important aspect to study. To study the performance of the tool as the number of queries increases, we analyzed SVT1 for varying number of queries. The running times along with the number of queries and the value for c is shown in Table 8.3. The table shows that the tool can handle a reasonable number of queries - given that most algorithms have a counterexample with three queries, our tool can handle four queries in a couple of hours.

In all the experiments so far, the value of ϵ was not fixed. So DiPC had to either prove privacy for all ϵ or find an ϵ where privacy is violated. Many automated tools are designed only to disprove differential privacy for a fixed ϵ. Our tool also has the capability to check differential privacy for a fixed value of epsilon. We tried the performance of the tool on SVT1 for a fixed ϵ. The results are reported in Table 8.4. As can be seen by comparing the numbers in Tables 8.3 and 8.4, fixing ϵ makes the problem easier to handle. On an average,

|Q|	c	Runtime (T1/T2)
1	1	0s/16s
2	1	0s/113s
2	2	0s/155s
3	1	0s/825s
3	2	0s/1202s
4	1	0s/4727s
4	2	0s/6715s

Table 8.3: Runtimes of SVT1 over different query length and counts, searching over all adjacency pairs and all $\epsilon > 0$, with parameters being [Δ=1, Domain={-1,0,1}]

the runtime seems to reduce by a factor of 2. This doesn't seem to be a significant gain in speed, given that we are giving up the power to search over a range of infinite values. Also, as one would expect, changing the value of ϵ doesn't seem to affect the runtime much.

|Q|	c	ϵ	Fixed ϵ Runtime (T1/T2)	General Runtime (T1/T2)
1	1	1.0	0s/7s	0s/16s
1	1	0.5	0s/8s	0s/16s
2	1	1.0	0s/43s	0s/113s
2	1	0.5	0s/46s	0s/113s
2	2	1.0	0s/95s	0s/155s
2	2	0.5	0s/113s	0s/155s
3	1	1.0	0s/307s	0s/825s
3	1	0.5	0s/265s	0s/825s
3	2	1.0	0s/541s	0s/1202s
3	2	0.5	0s/572s	0s/1202s
4	1	1.0	0s/1772s	0s/4727s
4	1	0.5	0s/1832s	0s/4727s
4	2	1.0	1s/2904s	0s/6715s
4	2	0.5	1s/3295s	0s/6715s

Table 8.4: Runtimes of SVT1 over different query length and counts, searching over all adjacency pairs and fixed ϵ, with parameters being [Δ=1, Domain={-1,0,1}]

Another common technique adopted by existing tools to speed up the runtimes is to check for a fixed input adjacency pair. DiPC has the capability to perform this optimization too. In Table 8.5, we have the results when a non differentially private algorithm, namely SVT3 was run with a single adjacency pair ([00...]$\sim$[11...]), while varying number of queries. This optimization makes a huge difference in the size of the generated Mathematica® script. We

notice that the running times is significantly lower in this case. As the number of queries increase, while the general runtime seems to increase by a factor of 10, the single pair runtime only doubles itself. Another interesting observation is that the time taken for 5 queries is lower than the time for 4 queries. This is because with 5 queries, the tool successfully finds a counterexample and terminates before checking the remaining inequalities.

#Queries	1 Pair Runtime (T1/T2)	General Runtime (T1/T2)	ϵ-Diff. Private
1	0s/15s	0s/25s	✓
2	0s/40s	0s/192s	✓
3	0s/100s	0s/1562s	✓
4	0s/199s	1s/10515s	✓
5	0s/141s	18s/5042s	✗

Table 8.5: Runtimes of SVT3 over different query lengths, searching over a single adjacency pair ([00...]∼[11...]) and all $\epsilon > 0$, with parameters being [c=1, Δ=1, Domain=$\{-1, 0, 1\}$, seq=(0<1)]

Additionally, DiPC can also verify (ϵ, δ)-differential privacy, which is a limitation of most of the existing tools. Algorithm Sparse (taken from [27]), presented in Figure 8.6 was used to evaluate DiPC's performance in this case. When $c = 1$, this algorithm is identical to Algorithm SVT1, where parameters c and Δ are replaced by parameter σ. This algorithm is, therefore, ϵ-differentially private. Further, our tool proves that the algorithm is not $\frac{\epsilon}{2}$-differentially private. The tool also shows that the algorithm is $(\frac{\epsilon}{2}, \delta = e^{-1/32})$-differentially private for $c = 1$. Additionally, we get a counterexample for $(\frac{\epsilon}{2}, e^{-4})$-differentially privacy.

When $c = 2$, Sparse differs from SVT1 since in this case we also need to choose r_T again after outputting a $\top$. The resulting program is $(\frac{\epsilon}{2}, \delta = e^{-1/64})$-differentially private. This is confirmed by DiPC. Further, DiPC also demonstrates that Sparse is not $(\frac{\epsilon}{2}, \delta)$-differentially private for $\delta = e^{-4}$. To the best of our knowledge, our approach is the first method to automatically check this.

A summary of various runs of the Sparse algorithm with different parameters have been presented in Table 8.6. We notice in the table, that the runtime by Approach 4 (Sec. 7.1.4 is significantly higher for $(\frac{\epsilon}{2}, e^{-2})$-Differential Privacy check. This is because the algorithm requires to perform inequality checks for every possible subset of outputs. Further, the runtime is much higher than what we would expect after accounting for the output subset probability comparisons and we suspect this might be due to memory overflows within Mathematica®.

We also see that Approach 5 (Sec. 7.1.5), improves the runtime significantly. However,

|Q|	c	δ	Output	Input-1	Input-2	ϵ	Approach 4 Runtime (T1/T2)	Approach 5 Runtime (T1/T2)
3	1	0	$[\perp\perp\top]$	[0 0 0]	[0 0 1]	1.17×10^{-16}	0s/55s	0s/47s
3	1	$e^{-1/32}$	$(\frac{\epsilon}{2}, \delta)$-Diff. Private				0s/256s	0s/167s
3	1	e^{-4}	$[\perp\perp\top]$	[0 0 1]	[1 1 0]	99/5	0s/211s	0s/148s
3	2	0	$[\top\perp\top]$	[0 0 0]	[0 0 1]	7.54×10^{-17}	0s/93s	0s/72s
3	2	e^{-2}	$(\frac{\epsilon}{2}, \delta)$-Diff. Private				0s/44882s	0s/289s
3	2	e^{-4}	$[\perp\perp\top]$	[0 0 1]	[1 1 0]	43/5	0s/659s	0s/181s

Table 8.6: Runtimes and Counterexamples of Sparse for $(\frac{\epsilon}{2}, \delta)$-Differential Privacy using Approach 4 and Approach 5 in Sec. 7.1 over different counts and delta values, searching over all adjacency pairs and all $\epsilon > 0$ and Domain=$\{0,1\}$

this approach has the caveat that it can only generate the input pair of the counterexample and not the output.

Interestingly, both the approaches are complementary to each other. We can use Approach 5 to generate a counterexample input pair and run Approach 4 with a single adjacency pair to generate the counterexample output.

REFERENCES

[1] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *IACR Theory of Cryptography Conference (TCC), New York, New York*, 2006. [Online]. Available: http://dx.doi.org/10.1007/11681878_14 pp. 265–284.

[2] G. Barthe, M. Gaboardi, J. Hsu, and B. C. Pierce, "Programming language techniques for differential privacy," *SIGLOG News*, vol. 3, no. 1, pp. 34–53, 2016. [Online]. Available: https://dl.acm.org/citation.cfm?id=2893591

[3] 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*, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863568

[4] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce, "Linear dependent types for differential privacy," in *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2429113 pp. 357–370.

[5] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin, "Probabilistic relational reasoning for differential privacy," *ACM Transactions on Programming Languages and Systems*, vol. 35, no. 3, p. 9, 2013. [Online]. Available: http://software.imdea.org/~bkoepf/papers/toplas13.pdf

[6] G. Barthe, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub, "Proving differential privacy via probabilistic couplings," in *IEEE Symposium on Logic in Computer Science (LICS), New York, New York*, 2016. [Online]. Available: http://arxiv.org/abs/1601.05047

[7] G. Barthe, N. Fong, M. Gaboardi, B. Grégoire, J. Hsu, and P. Strub, "Advanced probabilistic couplings for differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 55–67.

[8] D. Zhang and D. Kifer, "Lightdp: towards automating differential privacy proofs," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, G. Castagna and A. D. Gordon, Eds. ACM, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3009884 pp. 888–901.

[9] A. A. de Amorim, M. Gaboardi, J. Hsu, and S. Katsumata, "Metric semantics for probabilistic relational reasoning," *CoRR*, vol. abs/1807.05091, 2018. [Online]. Available: http://arxiv.org/abs/1807.05091

[10] A. Albarghouthi and J. Hsu, "Synthesizing coupling proofs of differential privacy," *PACMPL*, vol. 2, no. POPL, pp. 58:1–58:30, 2018. [Online]. Available: https://doi.org/10.1145/3158146

[11] Z. Ding, Y. Wang, G. Wang, D. Zhang, and D. Kifer, "Detecting violations of differential privacy," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243818 pp. 475–489.

[12] B. Bichsel, T. Gehr, D. Drachsler-Cohen, P. Tsankov, and M. T. Vechev, "Dp-finder: Finding differential privacy violations by sampling and optimization," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243863 pp. 508–524.

[13] S. McCallum and V. Weispfenning, "Deciding polynomial-transcendental problems," *Journal of Symbolic Computation*, vol. 47, no. 1, pp. 16–31, 2012.

[14] Anonymous, "Differential privacy," https://anonymous.4open.science/repository/febcbe47-1c53-41db-be91-ea98b4cf18c1/, 2018.

[15] D. Liu, B. Wang, and L. Zhang, "Model checking differentially private properties," in *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, ser. Lecture Notes in Computer Science, S. Ryu, Ed., vol. 11275. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-030-02768-1_21 pp. 394–414.

[16] D. Chistikov, A. S. Murawski, and D. Purser, "Bisimilarity distances for approximate differential privacy," in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 11138. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-030-01090-4_12 pp. 194–210.

[17] G. Barthe and B. Köpf, "Information-theoretic bounds for differentially private mechanisms," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 2011. [Online]. Available: https://doi.org/10.1109/CSF.2011.20 pp. 191–204.

[18] F. McSherry and K. Talwar, "Mechanism design via differential privacy," in *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*. IEEE Computer Society, 2007. [Online]. Available: https://doi.org/10.1109/FOCS.2007.41 pp. 94–103.

[19] C. Dwork, M. Naor, O. Reingold, G. N. Rothblum, and S. P. Vadhan, "On the complexity of differentially private data release: efficient algorithms and hardness results," in *ACM SIGACT Symposium on Theory of Computing (STOC), Bethesda, Maryland*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1536467 pp. 381–390.

[20] M. Lyu, D. Su, and N. Li, "Understanding the sparse vector technique for differential privacy," *Proceedings of VLDB*, vol. 10, no. 6, pp. 637–648, 2017, also appears as arXiv preprint arXiv:1603.01699. [Online]. Available: http://www.vldb.org/pvldb/vol10/p637-lyu.pdf

[21] M. Hardt and G. N. Rothblum, "A multiplicative weights mechanism for privacy-preserving data analysis," in *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, 2010. [Online]. Available: https://doi.org/10.1109/FOCS.2010.85 pp. 61–70.

[22] A. Gupta, A. Roth, and J. Ullman, "Iterative constructions and private data release," in *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Cramer, Ed., vol. 7194. Springer, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-28914-9_19 pp. 339–356.

[23] T.-H. H. Chan, E. Shi, and D. Song, "Private and continual release of statistics," *ACM Transactions on Information and System Security*, vol. 14, no. 3, p. 26, 2011. [Online]. Available: http://eprint.iacr.org/2010/076.pdf

[24] A. Gupta, K. Ligett, F. McSherry, A. Roth, and K. Talwar, "Differentially private combinatorial optimization," in *ACM–SIAM Symposium on Discrete Algorithms (SODA), Austin, Texas*, 2010. [Online]. Available: http://arxiv.org/pdf/0903.4510v2 pp. 1106–1125.

[25] D. Chistikov and C. Haase, "The Taming of the Semi-Linear Set," in *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, Eds., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6263 pp. 128:1–128:13.

[26] D. Berend and L. Bromberg, "Uniform decompositions of polytopes," *Applicationes Mathematicae*, vol. 33, pp. 243–252, 01 2006.

[27] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014. [Online]. Available: http://dx.doi.org/10.1561/0400000042