Implementing the Exponential Mechanism with Base-2 Differential Privacy

Christina Ilvento

Abstract

Despite excellent theoretical support, Differential Privacy (DP) can still be a challenge to implement in practice. In part, this challenge is due to the very real concerns associated with converting arbitrary or infinite-precision theoretical mechanisms to the often messy realities of floating point or fixed-precision. Beginning with the troubling result of Mironov demonstrating the security issues of using floating point for implementing the Laplace mechanism, there have been many reasonable concerns raised on the vulnerabilities of real-world implementations of DP.

In this work, we examine the practicalities of implementing the exponential mechanism of McSherry and Talwar. We demonstrate that naive or malicious implementations can result in catastrophic privacy failures. To address these problems, we show that the mechanism can be implemented exactly for a rich set of values of the privacy parameter $\varepsilon$ and utility functions with limited practical overhead in running time and minimal code complexity.

How do we achieve this result? We employ a simple trick of switching from base $e$ to base 2, allowing us to perform precise base 2 arithmetic. A short, precise expression is always available for $\varepsilon$, and the only approximation error we incur is the conversion of the base-2 privacy parameter back to base $e$ for reporting purposes. The core base 2 arithmetic of the mechanism can be simply and efficiently implemented using open-source high precision floating point libraries. Furthermore, the exact nature of the implementation lends itself to simple monitoring of correctness and proofs of privacy.

1 Introduction

As ever more data is collected about individuals for research or business purposes, ensuring that analyses of these data preserve individuals’ privacy becomes increasingly important. Differential Privacy [4], a theoretically rigorous framework for evaluating the privacy properties of mechanisms operating on private data, has emerged as the de facto standard for effective privacy protection. Differential privacy (DP) requires that the output of any computations over private data should be roughly indistinguishable from the output of those same computations if a single individual had changed her private data. For example, suppose a private database contains information about smoking behavior and demographic information for the patients in a particular health clinic. A differentially private computation over these data would yield the same insights regardless of whether any specific individual smoked. That is, the output of a DP computation over the true dataset $M(d)$ is roughly indistinguishable from the same computation over a neighboring database $M(d')$, in which one individual’s smoking behavior was different.

*Harvard John A Paulson School of Engineering and Applied Science. This work was supported in part by the Sloan Foundation.
The benefit of this type of definition is that every individual whose data is included in the database is guaranteed that their individual smoking behavior is not leaked by the outputs of the computation, but the computation can still gain valuable insights that hold in aggregate, e.g., whether a new smoking cessation program is effective. Unlike other proposals, e.g. \( k \)-anonymity, DP gives a privacy guarantee which is robust to subsequent post-processing and combination or composition with other private (or non-private) mechanisms and makes no assumptions about an adversary’s computational powers or side information. This makes DP a powerful tool for building privacy-preserving systems constructed from many separate privacy-preserving components.

Unfortunately, the practicalities of building computations or mechanisms which satisfy the requirements of DP can be challenging. In particular, many theoretically simple mechanisms which satisfy DP by adding noise drawn from a particular distribution are vulnerable to translation problems between the theoretical mechanism specification and the reality of limited precision arithmetic (as well as the complexities of timing and other side-channel problems). In particular, adding noise produced from inexact computations over finite precision values may contain artefacts leaking significant information about the original value. These translation problems which arise from leakage of information due to exposing the results of inexact calculations are well studied (e.g., \([10, 6, 1]\)).

New and important problems can arise even when the results of inexact computations are not revealed directly, in particular in the popular and highly flexible Exponential Mechanism of \([9]\). In this work, we examine the problems of implementing the exponential mechanism in practice. One might initially think that by revealing a result selected from a public list of possible outcomes (rather than the result of an inexact computation) that the exponential mechanism should be immune to these problems. We show two classes of attacks on a naive implementation of the exponential mechanism based on inexact floating-point arithmetic which allow an adversary to distinguish between two neighboring databases with high probability of success. We briefly discuss the difficulty of detecting and mitigating these attacks using inexact arithmetic. We propose a new view of implementing DP in practice by performing exact calculations by switching from base \( e \) to base 2 and show that this implementation technique gives strong privacy guarantees for practical implementations. We show how to translate the exponential mechanism into base 2, and give simple methods for expressing a wide range of privacy parameters and utility functions exactly. We give a reference implementation of the base-2 exponential mechanism using the GNU Multiple Precision Arithmetic Library via the \texttt{gmpy2} Python interface, and show the practical benefits of using exact arithmetic for auditing, monitoring correctness and giving proofs of privacy on the implementation itself. Finally, we briefly discuss extension of the base-2 DP technique to other settings and mechanisms.

1.1 Differential Privacy Preliminaries

Differential Privacy \([4]\) is a strong definition of privacy which requires that a mechanism operating on private data is stable. That is, the output of the mechanism cannot change “too much” if a single entry in the input database is changed. A key strength of DP is that it doesn’t apply to a single mechanism: any randomized mechanism satisfying the stability requirement is differentially private. We give the formal definition below.

**Definition 1.1** (Pure Differential Privacy). A randomized mechanism \( \mathcal{M} \) is \( \varepsilon \)-differentially private if for all adjacent databases \( d \sim d’ \), i.e. databases which differ in a single entry,

\[
\Pr[\mathcal{M}(d) \in C] \leq e^\varepsilon \Pr[\mathcal{M}(d’) \in C]
\]

where probability is taken over the randomness of the mechanism \( \mathcal{M} \).
Suppose we wanted to construct a DP mechanism to release a statistic of the database, $f(d)$. Several core DP mechanisms follow the template of computing $f(d) + \text{noise}$ scaled based on the sensitivity of $f$. The sensitivity of $f$ captures how much $f$ can change if a single entry of the database is changed, and is defined $\Delta f := \max_{d, d' \in D} |f(d) - f(d')|$. Naturally, computing a counting query, e.g. the number of individuals with a salary greater than $x$, has lower sensitivity than computing an average of a field, e.g. the average salary of all individuals in the database. This follows from observing that the number of individuals with a salary greater than $x$ can change by only 1 if a single individual’s value is changed. However, the average salary could change drastically if a single individual’s salary is changed. The best-known of these mechanisms is the Laplace Mechanism.

**Mechanism 1** (The Laplace Mechanism). Given a function $f : D \to \mathbb{R}$ with sensitivity $\Delta f := \max_{d, d' \in D} |f(d) - f(d')|$, a privacy parameter $\varepsilon$ and a database $d$, release $f(d) + \tau$ where $\tau$ is drawn from the sensitivity scaled Laplace distribution, given by probability density $\text{Lap}(t|\Delta f) = \frac{1}{2\Delta f} e^{-\frac{|t|}{\Delta f}}$. The Laplace Mechanism is $\varepsilon$-DP.

Notice that to implement this mechanism as written, one must sample from the Laplace distribution with arbitrary precision. This is problematic in practice as floating point cannot express every possible value in the distribution. In particular, taking $y = \ln(x)$ for uniformly distributed random floating point numbers $x$ (as one would to sample from the Laplace distribution) will result in gaps in the set of values for $y$. As shown by Mironov [10], this can be exploited by an adversary to determine $f(d)$ from $f(d) + \tau$ by examining the lowest order bits of the result. Roughly speaking, this attack takes advantage of the fact that the supports, i.e. the sets of all possible outcomes, of adjacent databases are different due to the differing artefacts of floating point computations. Fortunately, this particular attack on the Laplace mechanism (and similar concerns with other releases of noisy statistics under finite precision semantics) can be mitigated by careful implementation, as noted in [10]. We briefly highlight two other works also consider similar problems due to finite precision semantics. First, Gazeau, Miller and Palamidessi’s “Preserving differential privacy under finite-precision semantics” study the problem of bounding privacy loss due to finite-precision semantics [6]. Their approach is to use fixed-precision computation, and to allow for bounded privacy degradation due to error in scaled noise mechanism (e.g., Laplace). Balcer and Vadhan consider the question of efficiently constructing differentially private histograms under finite precision semantics [1]. They exhibit efficient mechanisms with accuracy competitive to the Laplace mechanism for counting queries and histograms. However, these solutions do not directly address issues faced in implementing the exponential mechanism.

The exponential mechanism, proposed by McSherry and Talwar [9], is a general purpose DP mechanism for releasing arbitrary strings, values or other outputs according to their private utility. For example, suppose a company is trying to set a price for a new piece of software. To set the price, they ask a set of potential buyers the price they are willing to pay, and they promise to keep the prices reported by each potential buyer private and to only release a DP estimate of the optimal price. The company now has a problem: computing the optimal price and then adding noise may result in a price that zero buyers are willing to pay. For example, if all buyers are willing to pay at most $10, setting a price of $10.01 will result in zero sales. Using the Laplace mechanism, there is in fact a 50% chance the company sets a price no buyers are willing to pay. The exponential mechanism handles this problem in an elegant way by assigning a utility score.

---

1. e.g., the CEO of a company may make more than all other employees combined so adding or removing such an individual could as much as double the average.
2. See section 4.1, [11] and [10] for more discussion of inverse transform sampling and the particulars of sampling from the Laplace distribution.
3. See [5] Section 3.4 for a more complete auction-related example.
(potential profit) based on the database (the maximum prices) to each potential outcome (each price in a pre-determined range) and then sampling a single outcome based on the utility scores. Using this mechanism, the price of $10 would have very high utility and thus a high probability of selection, whereas any price over $10 would have low utility and probability of selection.

**Mechanism 2** (Exponential Mechanism). Given a privacy parameter $\varepsilon$, an outcome set $O$ and a utility function $u : D \times O \to \mathbb{R}$ which maps (database, outcome) pairs to a real-valued utility, the exponential mechanism samples a single element from $O$ based on the probability distribution

$$p(o) := \frac{e^{-\varepsilon u(d,o)}}{\sum_{o' \in O} e^{-\varepsilon u(d,o')}}$$

If the sensitivity of $u$, $\Delta u := |u(d,o) - u(d',o)| \leq \alpha$ for all adjacent $d, d'$ and for all $o \in O$, then the exponential mechanism is $2\alpha\varepsilon$-DP.

By convention, the utility function is negated, and lower utility corresponds to higher probability. The exponential mechanism is useful in many settings in which the space of possible outcomes (e.g., integer values in a range, preset error strings, synthetic datasets, etc) is known a priori or the usefulness of a statistic is highly sensitive to noise (e.g., optimal price) but the sensitivity of utility is small. In fact, any DP mechanism can be expressed as an instance of the exponential mechanism by choosing the appropriate utility function (although it may not be the most computationally efficient method) [9]. For example, the Laplace mechanism is simply the exponential mechanism with utility function $u(f(d), o) = |f(d) - o|$ for outcome space $O = \mathbb{R}$.

An initial examination of the exponential mechanism might lead the reader to imagine that it is immune to the floating point issues of the Laplace mechanism, as no floating point calculation is released directly and the computations of $e^x$ shouldn’t have “too much” error. However, in the next section we will see that naive implementations of the exponential mechanism are extremely vulnerable to floating-point based attacks.

## 2 Subverting the Exponential Mechanism

To understand how a naive implementation of the exponential mechanism can be subverted, we first recall two important properties of floating point arithmetic. For the purposes of our examples, we assume 5 bits of precision for the mantissa and a maximum (minimum) exponent of $\pm t$. First, values which are too small to be expressed in the available bits of precision (i.e., smaller than $2^{-t}$) are rounded to zero. So it’s possible that the outcome of a multiplication of two positive numbers is zero, as shown below.

$$00001 \times 2^{-t} \times 00001 \times 2^{-t} = 0 \times 2^0$$

Second, adding two floating point numbers of different magnitudes can result in truncated values.

$$\begin{array}{c}
11111 \quad \times 2^t \\
+ 00001 \quad \times 2^{-t} \\
\hline
11111 \quad \times 2^t
\end{array}$$

4
The naive exponential mechanism using numpy

```python
import numpy as np

def naive_exp_mech(eps, u, O):
    weights = [np.exp(-(eps/2)*u(o)) for o in O]  # compute the weight of each element in the outcome space
    total_weight = sum(weights)
    c_weights = [sum(weights[:i]/total_weight) for i in range(0, len(O))]  # compute the cumulative weight of each element
    index = np.random.rand()  # sample a random value in [0,1].
    for i in range(0, len(O)):
        if c_weights[i] >= index:
            return O[i]  # return the element corresponding to the random index
```

Figure 1: Naive reference implementation of the exponential mechanism in Python.

Notice that the larger number $11111 \times 2^{44}$ needs all five bits of precision to express its value, so adding any value smaller than $1 \times 2^{44}$ cannot be accounted for. Thus even though a positive number $(1 \times 2^{-1})$ was added, the result is equivalent to the original value.

We now translate these two observations into two attacks on a naive Python implementation of the exponential mechanism, outlined in Figure 1. We assume that all utility functions are pre-screened to ensure that they satisfy sensitivity constraints, and all of our malicious utility functions will have sensitivity $\leq 1$.

**Attack 1 (Zero-rounding).** Suppose the attacker wishes to know if Alice is in the database. The attacker designs an outcome space $O = [k]$ and a utility function such that if Alice is in the database $u(o_1) = x$ and $u(o_{i>1}) = x + 1$. If Alice is not in the database $u(o_i) = x$ for all $i \in [k]$. Notice that the sensitivity of $u$ is 1 regardless of the choice of $x$. The attacker then sets $x$ such that $\text{np.exp}(-\frac{\text{eps}}{2} \cdot x) > 0$ but $\text{np.exp}(-\frac{\text{eps}}{2} \cdot (x + 1)) = 0$. Notice that in the case where Alice is in the database, the only viable outcome is $o_1$, but in the case where Alice is not in the database, all elements have equal (and positive) weight, so a random element will be selected. Thus, the attacker has distinguishing probability $\frac{k-1}{k}$ on a single run of the mechanism.

To mitigate Attack 1, one might suggest that we add an assertion that each weight is positive. However, this fix can also be subverted by taking advantage of truncated addition.

**Attack 2 (Truncated addition).** Knowing that small, but positive, weights are truncated in addition, the adversary constructs two utilities $x_1$ and $x_2$ and an outcome space $O = [k]$ such that

$$\text{np.exp}(-\frac{\text{eps}}{2} \cdot x_1) + \sum_{i \in [k]} \text{np.exp}(-\frac{\text{eps}}{2} \cdot (x_2 + 1)) = \text{np.exp}(-\frac{\text{eps}}{2} \cdot x_1)$$

but

$$\text{np.exp}(-\frac{\text{eps}}{2} \cdot (x_1 + 1)) + \sum_{i \in [k]} \text{np.exp}(-\frac{\text{eps}}{2} \cdot (x_2)) \approx 2 \cdot \text{np.exp}(-\frac{\text{eps}}{2} \cdot (x_1 + 1))$$

---

4 By convention $[n]$ indicates the set of integers $\{1, 2, \ldots, n\}$. For outcome spaces, we index the elements of $O$ as $o_{i \in [O]}$.

5 Expressions $\text{np.exp}$, etc indicate the evaluation of a particular implementation, whereas expressions $e^x$ indicate infinite precision unless otherwise noted.
and assigns utility $u(o_1) = x_l + 1$ and assigns $u(o_{i>1}) = x_s$ if Alice is in the database and $u(o_1) = x_l$ and $u(o_{i>1}) = x_s + 1$ if Alice is not in the database. As before, the sensitivity of $u$ is 1 regardless of the choice of $x_s$ and $x_l$. Notice that if Alice is in the database, the only viable outcome is $o_1$, but if she is not, then $o_1$ is chosen only about half of the time. (Note: of course the adversary could tip the scales further in their favor by increasing $k$.)

The troubling property of Attack 2 is that all weights are positive, and the attack can be adapted to ensure that all weights exceed some minimum positive value. Please see the accompanying github repository for practical demonstrations of these attacks. Indeed, one would need to monitor the outcome of each addition in the mechanism to ensure that values are not excessively truncated to try to catch such subversion. However, such monitoring is non-trivial as the additions are not expected to be exactly correct as $e^x$ cannot be expressed exactly in a finite number of bits. One approach is to clamp the allowed utilities to a range in which we can compute additions of the smallest and largest possible weights safely. But how do we decide on such a range, and what is the impact of the inexact computations on privacy (given the vagaries of floating point division as well as addition and approximate computations of $\text{np.exp}$)?

The key problem with declaring a certain range of utilities to be safe is the difficulty in characterizing the privacy loss due to inexact arithmetic. For example, suppose that a range was chosen which has a maximum error on the order of $2^{-k}$. If each individual weight and the total weights are on the order of $2^0$, this error may be negligible from a privacy perspective, but if each individual weight is on the order of $2^{-k}$, it is not. Thus any general guarantee we might give on the privacy loss of the implementation must be pessimistic. Furthermore, the more involved and complete the defenses for such attacks become, the more complicated and difficult to audit the code becomes, increasing the likelihood of errors and those errors being missed under audit.\footnote{At the risk of belaboring the point, we also note that the number of bits of precision for built in types in certain languages can be set by the compiler further increasing the difficulty of effective auditing.}

Finally, more nuanced instances of the exponential mechanism may explicitly rely on sampling from the exact exponential mechanism distribution. For example, suppose that the outcome set is modified based on private information and the proof of privacy relies on the distributions of the exponential mechanism over these outcome spaces are close, i.e., the difference in the supports has very low weight. Significant error or clamping of small values may break the proof of privacy, as the difference between the distributions may be increased significantly by increasing the relative weight of low probability values. The concept of $\delta-$negligibility introduced by Blocki, Datta and Bonneau in [2] is a practical example of a use of the exponential mechanism which relies on the exact decay in probabilities of low utility elements.

## 3 Implementing the Exponential Mechanism Exactly

Given these difficulties, we devote the remainder of this work to a simple, concise, exact implementation of the exponential mechanism which is free from floating-point exploits. The primary technical observation motivating our solution is that exact base 2 exponentiation and arithmetic is much easier than base $e$ for computers. Of course, there is more subtlety to the solution than simply performing calculations in base 2, and there are four technical contributions underpinning our results:

1. Base-2 DP: we define base-2 DP and show the simple relationship between base-2 DP and standard base-$e$ DP. We prove that the base-2 exponential mechanism gives an exact privacy
guarantee when implemented correctly, i.e., that there is no privacy penalty due to the specifics of the implementation.

2. Choosing privacy parameters and handling non-integer utility functions: we show how to choose a rich set of privacy parameters $\eta \propto \varepsilon$ such that $2^{-\eta}$ can be computed exactly with limited precision. We demonstrate how to handle utility functions with arbitrary precision if $2^{-\eta u}$ cannot be computed exactly, e.g. $u = \frac{1}{3}$, via randomized rounding. The key insight is that randomized rounding incurs no privacy penalty from inexact implementation as the proof of privacy follows from worst-case rounding behavior.

3. Sampling from normalized probabilities without division: we give two methods to handle sampling from normalized probabilities without using division. This allows us to sample from the exact distribution of the exponential mechanism with limited randomness.

4. Implementation: we give a reference implementation of the base-2 exponential mechanism, and demonstrate how exact arithmetic lends itself well to efficient, simple monitoring of correctness (and thus privacy) using the gmpy2 [8] Python interface to the GNU Multiple Precision Arithmetic Library [7]. We also demonstrate that our mechanism is comparably efficient to a naive implementation, and examine the properties of sampling from the Laplace distribution using our method.

3.1 Base-2 differential privacy

The motivation for base-2 differential privacy follows from a simple observation: computers are very good at exact binary arithmetic. Instead of computing $e^x$, we will compute $2^y$ for appropriately chosen $y$ to achieve the same result from a privacy perspective while still being able to take advantage of floating point arithmetic. We now formally define base-2 differential privacy.

**Definition 3.1** ($|_2$ Differential Privacy). A randomized mechanism $M$ is $\eta|_2$-differentially private if for all adjacent databases $d$ and $d'$,

$$
\Pr[M(d) \in C] \leq 2^{\eta} \Pr[M(d') \in C]
$$

where probability is taken over the randomness of $M$.

A very simple change of base proves the relationship between base-2 and base-$e$.

**Lemma 3.2.** Any mechanism which is $\eta|_2$-differentially private is $\ln(2)\eta$−differentially private.

**Proof.**

$$
\Pr[M(d) \in C] \leq 2^{\eta} \Pr[M(d') \in C] 2^{\eta} = e^{\ln(2)\eta}, \text{ thus } \Pr[M(d) \in C] \leq e^{\ln(2)\eta} \Pr[M(d') \in C].
$$

We can also re-state the exponential mechanism in base 2.

**Mechanism 3** ($|_2$ Exponential Mechanism). Given a privacy parameter $\eta$, an outcome set $O$ and a utility function $u : D \times O \to \mathbb{R}$ which maps (database, outcome) pairs to a real-valued utility, the $|_2$ exponential mechanism samples a single element from $O$ based on the probability distribution

$$
p(o) := \frac{2^{-\eta u(d,o)}}{\sum_{o \in O} 2^{-\eta u(d,o)}}
$$

If $|u(d,o) − u(d',o)| \leq \alpha$ for all adjacent $d, d'$ and for all $o \in O$, then the base-2 exponential mechanism is $2\alpha\eta|_2$-DP.
The beauty of base-2 DP and the base-2 exponential mechanism is that if we can implement the mechanism exactly, then we have an exact guarantee on the privacy properties of the mechanism as implemented:

**Theorem 3.3** (Exact implementation). Given a privacy parameter $\eta$, an outcome set $O$ and a utility function $u: D \times O \to \mathbb{R}$ such that $|u(d,o) - u(d',o)| \leq \alpha$ for all adjacent $d$ and $d'$ and all $o \in O$, any exact and correct implementation of the base-2 exponential mechanism parameterized with an exact implementation of $u$ is $2\alpha\eta|_{2-\text{DP}}$ and $2\alpha \ln(2)|_{2-\text{DP}}$.

Notice that in the case where $\eta$ and $u(d,o)$ are integers, that the base-2 exponential mechanism can be implemented exactly in floating point so long as the floating point computations have sufficient bits to store the entirety of the intermediate results.\(^7\) Thus, if we were willing to use privacy parameters $\varepsilon = 2c\ln(2)$ for positive integers $c$, and restricted ourselves to integer utility functions, implementation is as simple as choosing a good arbitrary-precision floating point library. In the next section, we show how to support smaller privacy parameters and use non-integer utility functions.

### 3.2 Non-integer privacy parameter and utilities

To handle a larger set of privacy parameters and non-integer utilities, we first address selection of privacy parameters assuming integer utilities, and then we extend our results to non-integer utilities. Throughout this section, we use $\lceil x \rceil$ to indicate rounding $x$ up to the nearest integer, and $\lfloor x \rfloor$ to indicate rounding down. $b_x := \lceil \log_2(x) \rceil$ is used to indicate the number of bits required to represent $x$ exactly in binary.

#### 3.2.1 Expressive privacy parameters

Recall that to run the base-2 exponential mechanism, we need to compute values of $2^{-\eta u} = 2^{-\varepsilon u}$. To handle values of $\eta$ other than 1 so that we can support a range of privacy parameter choices, the main consideration is the ease of computing $(2^{-\eta})^u$ for integer $u$. Notice that if $2^{-\eta}$ can be computed exactly, then $(2^{-\eta})^u$ can be computed by repeated squaring. First, we characterize a large set of $\eta$ which can be computed exactly, and then we show that the number of bits of precision needed to express $2^{-\eta}$ is not too large in this set. These two ingredients are sufficient to show that we can compute the weights for the base-2 exponential mechanism exactly for a wide range of privacy parameters with reasonable precision (for integer utility functions).

Consider privacy parameters of the form

$$\eta = -z \log_2(\frac{x}{2^y})$$

for integer $x$, $y$ and $z$ such that $\frac{x}{2^y} \leq 1$. Notice that $2^{-\eta} = (\frac{x}{2^y})^z$. To compute $x2^{-y}$ we need to perform a single exponentiation and a single multiplication, and to compute $(x2^{-y})^z$ we perform a second exponentiation. Notice that even for $z = 1$, for small values of $y$, we can still achieve a large set of $\varepsilon$. For example, $-\log_2(15/2^4) \approx 0.06$, $-\log_2(31/2^5) \approx 0.03$, etc.\(^8\)

The primary consideration for the choice of $x$, $y$, and $z$ is the bits of precision required to maintain the exact solution for $\eta$ and $2^{-\eta}$. The following simple lemma gives the relationship between the size of $x$ and $y$ and the size needed to store $y^x$.

---

\(^7\)This is not entirely trivial for division, but the workaround is simple and addressed in Section 3.4.

\(^8\)Balcer and Vadhan use a similar technique in setting a privacy parameter $\varepsilon'$ which is close to their desired parameter $\varepsilon$, but behaves well in the exponent in order to do efficient inverse transform sampling. Please see Theorem 4.7 and Algorithm 4.8 of [1].
Lemma 3.4 (Precision of exponentiation). Given positive integer inputs $x$ and $y$, $y^x$ can be expressed exactly in $\sum_{i \in [b_x]} 1_i b^2_i \leq b_y x$ bits, where $1_i$ is the indicator for whether the $i^{th}$ bit of $x$ is 1.

Proof. Recall that one can compute $y^x$ by repeated squaring by computing $\prod_{i \in [b_x]} (y^{2^i} 1_i + (1 - 1_i))$. Observe that computing $y^{2^i}$ involves squaring $y$ $i$ times, and each square doubles the number of bits of precision. Thus computing $y^{2^i}$ requires $2^i b y$ bits of precision. Multiplying a pair of numbers requires at most the sum of the bits of precision for each number and multiplying by 1 results in no increase in precision, so the total bits of precision required are at most $\sum_{i \in [b_x]} 1_i 2^i b_y$. As $\sum_{i \in [b_x]} 1_i 2^i = x$, the inequality follows. \qed

Thus we can conclude that to compute $\eta$ and $2^{-\eta}$ we require at most $(y + b_x) z$ bits, and for any integer $u$, computing $(2^{-\eta})^u$ will require $u z (y + b_x)$ bits of precision. As shown above, setting $z = 1$ and choosing small $y$ still allows for a wide range of $\eta$, so in practice the magnitude of $u$ is the primary concern. More formally, we state the following corollary detailing the required size of $y$ for a desired approximation of a particular choice of $\epsilon$.

Corollary 3.4.1. Given a desired privacy parameter $\epsilon$, choosing $z = 1$ and $y \geq -\ln(2) (\log_2(c) - \epsilon)$, there exists a choice $\eta$ of the form $\eta = -z \log_2 \left( \frac{x}{2^y} \right)$ which achieves additive approximation error of less than $\log_2(1 - c)$ for $\epsilon$ (for constant $c \in (0, 1]$).

Thus, the main consideration for controlling precision are the range of utilities. Controlling the magnitude of $u$ is straightforward given a pre-determined range of acceptable $u$ and clamping any observed utilities to this range. As long as the range is determined independent of the private information, clamping to the range has no impact on the privacy guarantee. We state without proof the following simple proposition that the exponential mechanism with clamped utility is DP:

Proposition 3.5. Given a utility function $u$ such that $\Delta u \leq \alpha$, clamp $(u, A, B)$ where $\text{clamp}(u, A, B)(x, o) := \min(\max(A, u(x, o)), B)$ has sensitivity $\Delta \text{clamp}(u, A, B) \leq \Delta u$.

The proof of the proposition follows from observing that clamping values cannot increase the difference in utility of adjacent databases.

3.2.2 Non-integer utilities

Many applications of the exponential mechanism are amenable to integer utility functions, e.g. converting decimal dollar values to cent values. However, there may be cases where the utility function provided (for example, by a third party) cannot be modified directly, or we wish to simulate a distribution (such as the Laplace distribution). We now consider the case of a utility function which provides arbitrary precision non-integer utilities.

The simple solution to non-integer utilities is to use randomized rounding, i.e. rounding up or down to the nearest integer with probabilities proportional to how close the value is to each. Our basic strategy is to show that randomized rounding incurs no privacy penalty from inexact implementation.

Definition 3.6 (Randomized Rounding Exponential Mechanism). Given a privacy parameter $\epsilon$, an outcome set $O$ and a utility function $u : D \times O \rightarrow \mathbb{R}$ which maps (database, outcome) pairs to a

---

9 The details of the corollary appear in the appendix.
real-valued utility, the randomized rounding exponential mechanism first assigns an integer proxy
utility
\[ p(u(d, o)) := \begin{cases} 
\lfloor u(d, o) \rfloor & \text{with probability } |u(d, o) - \lfloor u(d, o) \rfloor| \\
\lceil u(d, o) \rceil & \text{with probability } |u(d, o) - \lceil u(d, o) \rceil| 
\end{cases} \]

exponential mechanism samples a single element from \( O \) based on the probability distribution
\[ p(o) := \frac{e^{-\varepsilon p(u(d, o))}}{\sum_{o \in O} e^{-\varepsilon p(u(d, o))}} \]

We now consider the privacy of randomized rounding in the exponential mechanism. The argument follows from considering the worst possible set of rounding choices, and arguing that privacy loss is no worse than in that case.

**Lemma 3.7** (Privacy of arbitrary precision randomized rounding). Given an implementation of a utility function \( u : D \times O \to \mathbb{R} \) which guarantees that for any pair of adjacent databases \( |u(d', o) - u(d, o)| \leq \alpha \) for integer \( \alpha \) as implemented, the exponential mechanism with a randomized rounding function of arbitrary precision is \( 2\alpha \varepsilon - DP \).

**Proof.** Recall that given a utility function \( u \) such that \( |u(d, o) - u(d', o)| \leq \alpha \) the exponential mechanism is \( 2\alpha \varepsilon - DP \). Consider the composed utility function \( u'(d, o) := p(u(d, o)) \). Notice that if \( \Delta u' \leq \alpha \), then the randomized rounding exponential mechanism with utility function \( u' \) is \( 2\alpha \varepsilon - DP \).

Suppose to implement randomized rounding that we draw a number \( s \) uniformly at random from \([0, 1)\), and round up if \( s \leq |u(d, o) - |u(d, o)|| \) (and otherwise round down). Fix a particular choice of \( s \). Consider a pair of neighboring databases \( d \) and \( d' \) such that \( u(d', o) > u(d, o) \). Notice that it is impossible for the rounding procedure to result in a difference in composed utility of more than \( \alpha \). This follows from observing that there are two cases: either \( |u(d', o) - |u(d, o)\rfloor| < \alpha \), in which case any rounding results in a difference at most \( \alpha \), or \( |u(d', o) - |u(d, o)\rceil| = \alpha \).

In the latter case, notice that \( |u(d', o) - |u(d', o)\rceil| \leq |u(d, o) - |u(d, o)\rfloor| \) as \( \alpha - 1 < u(d') - u(d, o) \leq \alpha \). Therefore \( s \) is in one of three regions:

1. \( s \in [0, |u(d', o) - |u(d, o)\rfloor|] \), which results in both values rounded up,
2. \( s \in (|u(d', o) - |u(d, o)\rceil|, |u(d, o) - |u(d, o)\rfloor|] \), which results in \( u(d', o) \) rounded down and \( u(d, o) \) rounded up, or
3. \( s \in (|u(d, o) - |u(d, o)\rfloor, 1) \), which results in both rounded down.

Thus, for any \( s \) rounding never results in a difference between \( u(d, o) \) and \( u(d', o) \) greater than \( \alpha \). (The symmetric argument follows for any \( o \) such that \( |u(d, o) - u(d', o)| \).

Thus, for any \( s \), \( \Delta u'|s \leq \alpha \). Therefore \( \Delta u' \leq \alpha \) as the probability of drawing any particular \( s \) is independent of the database. Notice that the privacy proof does not rely on the precision of \( s \) or the degree to which \( s \) is sampled perfectly from the uniform distribution over \([0, 1)\), only that the choice of \( s \) is independent of \( d \). In fact, we could fix a single \( s \) and use it for every instance of rounding, and the proof of privacy would still hold.

\( \square \)

Although randomized rounding has a straightforward proof of privacy, it’s not immediately obvious how well the exponential mechanism with randomized rounding approximates the original non-integer utilities. Unfortunately, characterizing the expected weight for a given element in the outcome space is complicated by the large space of possible normalization factors which depend on
the rounding behavior of the utilities for other elements. Fortunately, we can derive a bound for checking the difference in distributions over outcomes between the randomized rounding exponential mechanism, and the original (unrounded) mechanism. If this difference is small, we can be confident that we are sampling from a distribution very close to the originally intended distribution.

Informally, we derive this bound by lower-bounding the probability that the randomized rounding mechanism outputs a particular value, and upper bounding by computing the maximum point-wise error on the lower bound. More concretely, we first determine a lower bound on the probability that an element \( o \) is output by using the fact that \( E[X] \geq \frac{1}{E[1/X]} \) for positive random variable \( X \). We then compute the total probability mass assigned by the lower bound for all of the outputs in \( O \) to determine it’s maximum error, which is concentrated on a single point in the worst case. By adding this point-wise worst-case to the lower bound, we can give a matching upper bound. We can then take the worst-case error between either the upper or lower bound and the original probability. Note that as the width between the upper and lower bounds decreases, the error also decreases. We state the bounds more formally in the lemma below:

**Lemma 3.8.** Given a privacy parameter \( \epsilon \), a utility function \( u \) and an outcome set \( O \), the difference in the probability of outputting an outcome \( o^* \in O \) between the randomized rounding exponential mechanism and the original exponential mechanism is upper-bounded by \( A + |B| \) where

\[
\begin{align*}
p_{u(d,o)} &= |u(d,o) - \lceil u(d,o) \rceil| \\
q_{u(d,o)} &= \frac{p_{u(d,o)} e^{-\epsilon |u(d,o)|}}{e^{-\epsilon |u(d,o)|} + \mathbb{E}\left[\sum_{o' \in O \setminus \{o\}} e^{-\epsilon u(d,o')}\right]} + \frac{(1 - p_{u(d,o)}) e^{-\epsilon \lceil u(d,o) \rceil}}{e^{-\epsilon \lceil u(d,o) \rceil} + \mathbb{E}\left[\sum_{o' \in O \setminus \{o\}} e^{-\epsilon u(d,o')}\right]} \\
A &= 1 - \sum_{o \in O} q_{u(d,o)} \\
B &= \frac{e^{-\epsilon u(d,o^*)}}{\sum_{o \in O} e^{-\epsilon u(d,o)}} - q_{u(d,o^*)}
\end{align*}
\]

A complete proof of the Lemma is included in the Appendix. To build intuition, we give the following informal explanation of each part.

- \( p_{u(d,o)} \) is the probability that \( u(d,o) \) is rounded down.
- \( q_{u(d,o)} \) is a lower bound on the probability that the randomized rounding mechanism outputs \( o \). The first term is a lower bound on the probability conditioned on rounding down, and the second is conditioned on rounding up.
- \( A \) is the cumulative estimation error of the lower bound for all elements and therefore an upper bound on the estimation error for \( q_{u(d,o)} \) for any single element. Why? Each element’s probability is lower bounded by \( q_{u(d,o)} \), so the sum \( \sum_{o \in O} q_{u(d,o)} \) would be 1 if the estimates were exact. The maximum error on any given element’s probability is therefore given by subtracting the sum of the lower bounds from 1.
- \( B \) is the difference between the lower bound probability for the randomized rounding mechanism and the probability given by the original mechanism for outputting \( o^* \).

\( A + |B| \) is the sum of the error on the lower bound and the difference between the lower bound and the original unrounded mechanism probability. It is therefore the maximum difference between the probabilities of outputting \( o^* \) in the original unrounded mechanism and the randomized rounding mechanism.
The tightness of the error bound is directly related to the size of $A$, and as we will see in Section 4.1, the Lemma gives a tight estimate of the error in the important setting of implementing Laplace noise.

### 3.3 Normalized sampling without division

The final ingredient required to implement the exponential mechanism with exact arithmetic in base 2 is to demonstrate how to sample an outcome, i.e., how to normalize the weights. The challenge with division is that even if numerator and denominator can be expressed in a small number of bits, their quotient may not be (e.g., $\frac{11}{7}$). There are two alternatives. The first is to express each probability as a rational, and assign a contiguous region in $[0, 1)$ to each element based on its probability, and then sample a random rational value in $[0, 1)$ with sufficient precision, i.e., the greatest common multiple of the divisors, to select the element. The other alternative is to compute the total weight $t$ and assign each element a contiguous region in $[0, t]$ with length equivalent to its weight, (i.e., the first element is assigned $[0, w_1)$ the second $[w_1, w_1 + w_2)$ and so on).

Sampling a random value in $[0, t)$ is then identical to sampling from the normalized distribution. The key observation is that this normalized sampling mechanism only requires a random value with precision equal to the maximum precision of the computation of any individual weight or sum of weights. The second option turns out to be straightforward to implement, and requires on the order of precision random bits with high probability when all weights are small, and performs significantly better when there is variation in weights.

**Lemma 3.9** (Normalized sampling). Given a set of weights $W$ with total weight $t$ and a maximum precision $p$ such that the addition of any combination of $W$ can be expressed exactly with precision $p$, Algorithm 1 outputs the index of an element in $W$ with probability distribution identical to that of the normalized exponential mechanism and the procedure uses $O(p)$ random bits with high probability.

**Proof.** The proof consists of two parts: first, correctness of the distribution assuming sufficient precision; second, sufficiency of precision and bounding the number of random bits required.

**Correctness.** Notice that (assuming infinite precision) this procedure amounts to partitioning the range $[0, 2^k)$ between the elements of $W$ according to their weight (including a dummy value if needed), sampling a value $s$ in $[0, 2^k)$ and choosing an element in $W$ based on which partition $s$ lands in. If the dummy value is selected, the procedure restarts. To see the correctness of the sampling procedure, observe that in each iteration of the while loop, any elements “ruled out” by the bits of $s$ we have seen so far are removed. That is, any element that has been assigned a range strictly smaller than $s$ or strictly larger than $s + 2^j - 1$ is removed. This leaves the set of elements which still could be reached by the remaining bits of $s$, namely elements that have some portion of the range $[s, s + 2^j)$. The probability that any given element is sampled is equivalent to the probability that a random value $s \in [0, t)$ falls into its assigned range of $[0, t)$, thus, each element is sampled with probability $\frac{w_i}{t}$, which is equivalent to the exponential mechanism.

**Randomness required.** Notice that each iteration of the while loop at Line 14 requires at most $p$ random bits before a single value is identified. This follows from observing that at most one element can “claim” any range $[a2^{-p}, (a + 1)2^{-p})$ as all combinations of $W$ can be expressed in $p$ bits of precision. Thus it suffices to show that the loop only resets (i.e., the dummy value is selected, see Line 26) a constant number of times with high probability. Notice that the dummy value can only be selected if the first random value selected is 1, as the dummy value is only ever
Algorithm 1 The normalized weighted sampling algorithm

**Inputs:** $W$, a set of weights.

**Outputs:** $i$, an index sampled according to $p(i) := \frac{w_i}{\sum_{j \in W} w_j}$.

1: procedure NORMALIZEDSAMPLE($W$)

2: $t \leftarrow \sum_{w \in W} w$ \quad \triangleright \text{The total weight.}

3: for $i \in \{1, \ldots, |W|\}$ do \quad \triangleright \text{Compute the cumulative weight for each element.}

4: 

5: $c_i \leftarrow \sum_{j=1}^i w_j$ \quad \triangleright \text{Each element is assigned the region $[c_{i-1}, c_i)$.}

6: $k \leftarrow \lceil \log_2(t) \rceil$ \quad \triangleright \text{The smallest power of 2 at least as large as $t$.}

7: if $2^k > t$ then \quad \triangleright \text{Add an element ⊥ and append $2^k - t$ to the weights.}

8: $W \leftarrow W \cup \{⊥\}$

9: $c_{|W|} \leftarrow 2^k - t$ \quad \triangleright \text{Total weight is now $2^k$.}

10: $s \leftarrow 0$

11: $j \leftarrow k - 1$

12: while $|R| > 1$ do \quad \triangleright \text{Re-sample the remaining elements.}

13: $r_j \sim \text{Unif}(0, 1)$

14: $s \leftarrow s + r_j 2^j$

15: for $i \in R$ do

16: \hspace{1em} if $c_i \leq s$ then \quad \triangleright \text{s cannot be in $[c_{i-1}, c_i)$, even if all draws are 0.}

17: \hspace{2em} $R \leftarrow R \setminus \{i\}$

18: \hspace{1em} if $i > 0$ and $c_{i-1} \geq s + 2^j$ then \quad \triangleright \text{s cannot be in $[c_{i-1}, c_i)$, even if maximum value added.}

19: \hspace{2em} $R \leftarrow R \setminus \{i\}$

20: $j \leftarrow j - 1$

21: if $|R| = 1$ and ⊥ $\in R$ then \quad \triangleright \text{Restart if the dummy value is selected.}

22: $s \leftarrow 0$

23: $j \leftarrow k - 1$

24: $R \leftarrow [|W|]$

25: return $l$
assigned a range within $[t/2, t)$. \(^{10}\) The probability that $c$ samples all have 1 as their first bit is $2^{-c}$. Thus, with probability at least $1 - 2^{-c}$, $c$ iterations will result in a successful sample. \(\square\)

A benefit of this particular method for sampling without division is that we can sample very efficiently if there are many high weight elements. For example, suppose that an element $o_1$ has about half of the total weight. Then with probability about $\frac{1}{2}$ the sampling method will require only a single bit of randomness. However, it is important to note that this method can introduce a timing side-channel, in that making the weights more or less uniform can increase or decrease the amount of precision, and thus time, needed for sampling. For completeness, we present a second version of the sampling algorithm in Appendix A.1 with running time independent of the weight distribution (although it is demonstrably slower in practice).

### 3.4 Implementation

The reference implementation of the base-2 exponential mechanism is available on github. The implementation consists of approximately 200 lines of commented Python code.

#### 3.4.1 High precision arithmetic

Given Theorem 3.3, the goal of our implementation is to ensure that all arithmetic is exact. To that end, we use the GNU Multiple Precision Arithmetic Library (GMP) via the `gmpy2` Python interface, which allows us to perform high precision binary arithmetic and explicitly raise exceptions in cases of inexact results. The high precision types `mpz` (integers) and `mpfr` (reals) exposed by `gmpy2` are “drop-in” replacements for built-in `int` and `float`s, making the code easy to read. The library also provides a method for determining the maximum precision achievable on a given system at run-time, preventing any errors due to exceeding any system-specific thresholds and making the code easier to audit.

#### 3.4.2 Code structure and monitoring

Our reference implementation implements the exponential mechanism as a class with associated privacy and safety parameters. In order to have an up-front guarantee on the precision required, we fix a range of allowed utilities and privacy parameters, and ensure that the precision used for computations is large enough to accommodate these choices. Specifically, we include parameters for: maximum and minimum utility for clamping, maximum outcome space size, privacy parameters $x, y$ and $z$ determining $\eta = -z \log_2(\frac{x}{y})$.

When a new `ExpMech` object is instantiated, we first confirm that the parameters are of the appropriate types and then we set the precision by repeatedly trying to calculate the necessary weights and sums for the exponential mechanism and increasing the precision until either the maximum precision allowed by the system is exceeded or we can successfully complete these test computations. Algorithm 2 outlines the logic of these tests. We then store the precision and set the appropriate `context` parameters for `gmpy2` to ensure that any inexact arithmetic on `mpz` or `mpfr` values results in an exception. By setting the precision before observing the utility function or outcome space, we explicitly sequester any up-front precision problems from the private data, removing a potential side-channel.\(^{11}\) Once the `ExpMech` class is initialized, the caller sets a utility function, which is explicitly wrapped in randomized rounding logic. Finally, the mechanism logic

\(^{10}\)Strictly speaking, $s$ will also need at least one additional lower bit to be 1, but we give the simpler version as it’s sufficient for our purposes.

\(^{11}\)e.g., the adversary learning some property of the data because the maximum system precision is exceeded.
Algorithm 2 Minimum precision determination

**Inputs:** \( u_{\text{min}} \), the minimum utility, \( u_{\text{max}} \), the maximum utility (by convention \( u_{\text{min}} \) is the minimum weight utility, i.e., it may have larger magnitude than \( u_{\text{max}} \)), \( o_{\text{max}} \) the maximum number of outcomes, the privacy parameter \( \eta \), \( p_0 \) the initial precision.

**Outputs:** \( p \), a sufficient precision no more than twice the size of the minimum precision to successfully run the base-2 exponential mechanism.

1: procedure ComputePrecision\((u_{\text{min}}, u_{\text{max}}, o_{\text{max}}, \eta, p_0)\)  
2: \( p \leftarrow p_0 \)  
3: while CheckPrecision\((u_{\text{min}}, u_{\text{max}}, o_{\text{max}}, \eta, p)\) fails do  
4: \( p \leftarrow 2p \) \Comment{Double the precision until the required computations succeed.}  
5: return \( p \)

6: function CheckPrecision\((u_{\text{min}}, u_{\text{max}}, o_{\text{max}}, base, p)\)  
7: Set the precision to \( p \)  
8: Set arithmetic library to return failure on inexact arithmetic  
9: \( \text{maxsum} \leftarrow \sum_{i \in [o_{\text{max}}]} 2^{-\eta u_{\text{max}}} \)  
10: \( \text{minweight} \leftarrow 2^{-\eta u_{\text{min}}} \)  
11: \( \text{maxmin} \leftarrow \text{minweight} + \lceil \text{maxsum} \rceil \)

can be run using the rounded utility function. Throughout the mechanism logic, we explicitly check that no modifications have been made to the context parameters for gmpy2 (e.g., by a malicious utility function) to ensure that the appropriate precision is maintained and that inexact arithmetic results in exceptions.

**Logic.** The logic of our implementation is nearly identical to the naive implementation, but includes randomized rounding of the utility function, utility clamping (i.e., clamping utilities to the pre-specified range) as well as additional monitoring of exact arithmetic.

**Handling exceptions.** Under normal operation, we do not expect any exceptions to be raised due to inexact arithmetic. In the case of an exception, the system designer can either choose to retry with higher precision, or take the failure as an indication that debugging is needed. For example, the exception might be raised in the utility function itself and be entirely unrelated to floating point errors. We note that the problem of errors and exceptions being used as a side channel in DP is well known, and other techniques such as subsample and aggregate can provide privacy-preserving error handling for these cases. The critical point is that the exception raised is the appropriate signal to the database administrator that an error has been exploited either intentionally or unintentionally.

3.4.3 Performance

The main source of performance change in the exact mechanism versus the naive implementation is the setup cost, and the cost of using gmpy2 versus numpy or built-in functions. (For all tests, we used the un-optimized version of Algorithm 1, included in Appendix A.1). For high precision arithmetic, GMP is generally more performant than built-in types. As such, for bounded utilities these costs are limited, and a comparison of performance is included in Figure 4. In the case of bounded utilities, increasing the number of elements in the outcome space results in better performance from the base-2 mechanism than the naive reference. This is illustrated in Figure 4(a), which compares the running time of each implementation simulating the Laplace distribution.
# The Base-2 Exponential Mechanism Reference Implementation

```python
from gmpy2 import mpfr, mpz

class ExpMech:
    # Check that the appropriate precision and context settings are in place
    def check_context(self):
        ...
    # A method to test whether the current precision is sufficient for intended
    # usage.
    def check_precision(self):
        ... # See Algorithm 2
    # Initialize a new mechanism
    def __init__(self, rng, eta_x = 1, eta_y = 0, eta_z = 1,
                 u_min = 10, u_max = 0, max_O=100, \
                 min_sampling_precision = 10):
        ...
        # Set the gmpy2 library context to trap on inexact arithmetic, overflows, etc.
        ctx = gmpy2.get_context()
        ctx.trap_inexact = True
        ...
    # Set the rng, privacy parameters and utility bounds
    ...
    # Compute the required precision for the desired parameters
    ... # See Algorithm 2
    # Sample a random value with p bits of precision in [0,2^(start_pow+1))
    def get_random_value(self, start_pow, p):
        ... # See Algorithm 1
    # Set the utility function, wrapped in randomized rounding and clamp logic
    def set_utility(self, util):
        ...
    # Sample an index from W according to the normalized weight of each entry
    def binary_sample(self, W):
        ... # See Algorithm 1
    # Exact exponential mechanism
    # Inputs: O: the set of outputs
    def exact_exp_mech(self, O):
        ... # Check that O matches size requirements
        U = [self.u(o) for o in O] # Get utilities
        W = [pow(mpfr(self.base),mpfr(u)) for u in U] # Compute weights
        self.check_context()
        return O[self.binary_sample(W)] # Sample
```

Figure 2: An outline of the ExpMech class.
from exponential_mechanism.expmech import *

# Initialize outcomes and utilities
u = lambda x : abs(0 - x)

# privacy parameters
y = 1
x = 1

# outcome space
gamma = 2**(-5)
k = 256
O = [i*gamma for i in range (-k,k +1)] # [-8, 8] in 0.3125 increments

# random bit generator
rng = lambda : np.random.randint(0,2) # np.random is a placeholder for demonstration purposes only

# Initialize the mechanism
e = ExpMech(rng, eta_x =x, eta_y =y, eta_z =1) # use defaults for range bounds
e.set_utility(u) # set the utility function

# run the mechanism
result = e.exact_exp_mech(O)

Figure 3: Example usage of base-2 mechanism to sample from the range $[-8, 8]$ in increments of $2^{-5}$ for the utility function $u(o) = |0 - o|$.

with increasing granularity. On the other hand, when the utilities are allowed to increase along with the outcome space size, the performance of the base-2 mechanism degrades more quickly than the naive mechanism, as the naive mechanism does not ever increase the precision of its operations, as shown in Figure 4(b) and 4(c). However, as shown in Figure 4(c), the optimized sampling logic nearly meets the performance of the naive mechanism, as very few random bits are required to select from such a small sample space. In short, the optimized sampling logic provides significantly better performance for high precision settings, but it can open the door to explicit timing channel attacks, and must be used with caution.

4 Applications

The key benefit of the exponential mechanism is its universality: any utility function and any (finite) outcome space can be used. A natural question to ask is whether the base-2 implementation of the exponential mechanism can be used to efficiently implement interesting utility functions, in particular to implement the Laplace Mechanism. We now show how to apply the base-2 exponential mechanism to sample Laplace noise. There are three issues to consider: first, how to convert the infinite-outcome space Laplace Mechanism to a finite outcome space; second, how efficient the base-2 implementation is; and third, how much randomized rounding in the base-2 mechanism affects the outcome distribution. We’ll see that each of these issues can be resolved satisfactorily.

4.1 Choice of outcome space

Recall that the Laplace mechanism (Mechanism 1) computes a function $f$ of the database and then adds noise drawn from the Laplace distribution, described by probability density function

$$\text{Lap}(t|\Delta f) = \frac{e^{-|t|/\Delta f}}{2\Delta f}.$$ Notice that the range of possible noise values is infinite - although the probability of observing a sample $\geq 14$ is less than one in a million for $\varepsilon = 1$. The standard method
(a) Time in seconds for the discretized Laplace distribution with granularity $\gamma$.

(b) Time in seconds for outcome spaces of increasing size and utility range.

(c) Time in seconds for outcome spaces of fixed size ($k = 100$) and increasing utility range.

Figure 4: Comparisons of the time taken by the naive exponential mechanism versus the base-2 (un-optimized) implementation. 4(a) compares the performance when sampling from the clamped, discretized Laplace distribution with increasing granularity, i.e. $O = \{\gamma i, -\gamma i | \gamma i \in [-10, 10]\}$ and $u(o) = |o|$. The utility range is clamped from $[-10, 10]$, and the base-2 mechanism performs better for increasing granularity (smaller $\gamma$). 4(b) compares the performance for increasingly large sets of discrete outcomes, with increasing utility ranges, i.e. $O = [n]$ and $u(o) = x$. As the utility range increases, the performance of the (un-optimized) base-2 mechanism, which allocates additional precision, degrades compared with the naive mechanism. 4(c) compares the performance of the naive mechanism with the base-2 mechanism with and without sampling optimization for increasing utility ranges (requiring greater precision for the base-2 mechanism) with fixed outcome set size to illustrate the impact of increasing precision on running time. The optimized base-2 mechanism average time is nearly identical to the naive mechanism, as very little randomness is needed to sample from such a small outcome space. This clearly illustrates the impact of the required precision on sampling time. All points indicate an average of 10 trials run on Google’s Colaboratory free tier without GPU.
for sampling from the Laplace distribution is to use inverse transform sampling, i.e., to sample a uniform value in \( U \in [0, 1] \) and solve for \( \text{CDF}(t) = U \), where \( \text{CDF} \) refers to the cumulative density function of the Laplace distribution.\(^{12}\) In a simple implementation, this amounts to computing \( \tau = \frac{x}{\pi \Delta} \ln(1 - U) \) for a uniformly random values \( U \in [0, 1] \).\(^{13}\) As discussed extensively in [10], the danger of implementing this mechanism using standard (even highly accurate) implementations of \( \ln \) is that there are certain gaps in the low order bits of the values produced by \( \ln(x) \) for \( x \in [0, 1] \) even when \( x \) is drawn at regular intervals. These gaps result in differences in the support (possible outcomes) of the mechanism on adjacent databases, which can then be used to distinguish possible values of the original output of \( f \). By “snapping” \( f \) to a fixed set of intervals, these low order bits are safely eradicated.

However, the Laplace mechanism can also be implemented via the exponential mechanism by setting the utility function to be \( u(d, o) = |f(d) - o| \), although there is some subtlety in the selection of \( O \). Given our exact implementation of the finite base-2 exponential mechanism, we discuss methods for for implementing a base-2 Laplace mechanism, and their trade-offs, below.

**Exact privacy with noise bias.** The first option is to choose a range of outcomes \( O \) a priori and run the base-2 exponential mechanism with \( u(d, o) = |f(d) - o| \). The benefit of this implementation style is that the outcome space is set before ever seeing \( f \) or \( d \), and thus there is no possibility of leakage based on floating point arithmetic artefacts in the output space. Of course, this implementation is an approximation of the Laplace mechanism, because we cannot use an infinite outcome space. In particular, if the choice of \( O \) is skewed, the expected error of the mechanism may not be zero (as in an exact implementation of Laplace). For example, if \( \max(O) < f(d) \), then the error will be one-sided.\(^{14}\) However, the privacy of the mechanism is still exact as the sensitivity of \( f \) and thus \( u \) is bounded. In cases where the expected range of \( f \) is well-known, the exact privacy semantics may be worth the potential error bias.

**Centered noise with approximate privacy and mixed precision.** An alternative is to approximate noise from the zero-centered Laplace distribution (at fixed, symmetric intervals), and then add the result to a computation of \( f(d) \) snapped to the same interval. That is, choosing \( O = \{ 0 + \gamma x_i, 0 - \gamma x_i | i \in [k] \} \) for a fixed offset \( \gamma \). This method is approximately private, as there are certain outcomes (i.e., the smallest or largest elements in \( O \) which may exactly indicate which database gave rise to the result. However, the probability of seeing such an outcome is low if \( k \) is large enough, and can be quantified exactly. Thus, although the method is approximately private, the \( \delta \) can be computed exactly. Although this method has centered noise, it does present a problem of the mismatch of the precision of \( f(d) \) with the precision of \( O \), and thus care must be taken to appropriately “snap” \( f(d) \) to the required precision. Roughly speaking, the snapping mechanism, originally proposed in [10], takes a value \( f(d) \) and snaps it to a predetermined interval, i.e. \( \text{snap}(x, \gamma) = \arg \min_{\{y_i\in\mathbb{Z}\}} |x - \gamma i| \). (Further details on implementing the snapping mechanism can be found in [3].) Notice that if \( O \) has an interval width of \( \gamma \), and \( f(d) \) is snapped to the nearest \( \gamma \) interval, that there are no remaining low-order bits to leak the original value of \( f(d) \). Of course, if the snapping procedure can change the sensitivity of \( f \), this must be taken into account. The downside of this method is that new code must be introduced and properly integrated for

\(^{12}\)Recall that the cumulative density function is the integral of the probability density function from \(-\infty\) to \( t \), and in the case of Laplace is very simple to compute.

\(^{13}\)See [10] for a longer discussion of this technique for the Laplace mechanism in the context of DP and [11] for a more general discussion of sampling from the Laplace distribution.

\(^{14}\)Note that this skew is greater than the skew due to clamping the range alone, as clamping maps any non-zero probability outside the range to the range limit. In this case, no probability is assigned outside the range limits.
snapping $f(d)$ and ensuring that the result of $\text{snap}(f(d), \gamma) + \tau$ is exact, although [3] does provide an implementation of the snapping mechanism.

**Centered noise with approximate privacy or pessimistic exact privacy.** The spirit of the two approaches above can also be combined:

1. Choose interval $\gamma$ and range $[-B, B]$.
2. Compute $\hat{f}(d) = \text{snap}(f(d), \gamma)$.
3. Choose $O = \{\hat{f}(d) - \gamma i, \hat{f}(d) + \gamma i \mid i \in \lbrack B / \gamma \rbrack\}$.
4. Run the base-2 exponential mechanism with $u(o) = |\hat{f}(d) - o|$, adjusting sensitivity of $f$ if needed to account for $\text{snap}$.
5. Clamp the output of the mechanism to $[-B, B]$.

Notice that the support of the outcomes of mechanism is the same for any $f(d)$. The mechanism can be viewed as approximately private\(^{15}\) as in any pair of adjacent databases, each has an excess probability $\propto 2^{-2\eta B}$ on either $B$ or $-B$. That is, the difference in probabilities assigned to the boundaries will be slightly larger than allowed by pure $\varepsilon$-DP, although it is still bounded proportional to $2^{-2\eta B}$. Alternatively, one can derive a worst-case bound on the ratio of the probabilities assigned to outcomes outside the range $[-B, B]$. (As in the above, if snapping changes the sensitivity of $f$, this must be taken into account).

### 4.2 Efficiency and the impact of randomized rounding

The primary question with using the base-2 exponential mechanism implementation for the Laplace mechanism is how much the use of randomized rounding impacts the accuracy of the mechanism. In brief, randomized rounding has a limited impact and the rounded mechanism behaves very similarly to the unmodified, inexact mechanism in the case of simulating the Laplace distribution. Furthermore, a small outcome set size ($< 1,000$) is sufficient to closely approximate the true Laplace distribution.

Figure 5 illustrates the distribution of outputs simulating the Laplace distribution for the randomized rounding mechanism, the naive mechanism, and the `numpy` implementation of the Laplace distribution. Notice that the distribution is nearly identical in both cases. The Kolmogorov-Smirnov test statistic for the pairs of distributions is also small (0.02), indicating that there isn’t sufficient evidence to conclude that the distributions are different. Figures 6 and 7 illustrate how closely the lower and upper bounds given by Lemma 3.8 bound the target probability of the naive mechanism. In particular, the width of the bound is closely related to the granularity of the outcome space, and as granularity increases, the width of the bound decreases, giving smaller point-wise error bounds.

### 5 Discussion and future work

Floating-point issues in implementations of the Laplace mechanism and other additive noise mechanisms are well-known. In this work, we have demonstrated new floating point issues which arise

\(^{15}\)Formally, a mechanism is $(\varepsilon, \delta)$-DP or *approximately* DP if for all adjacent $d, d'$, $\Pr[M(d) \in C] \leq e^\varepsilon \Pr[M(d') \in C] + \delta$.\footnotetext[15]{Formally, a mechanism is $(\varepsilon, \delta)$-DP or *approximately* DP if for all adjacent $d, d'$, $\Pr[M(d) \in C] \leq e^\varepsilon \Pr[M(d') \in C] + \delta$.}
Figure 5: A comparison of the outputs of the randomized rounding base-2 exponential mechanism with the unmodified exponential mechanism (the naive mechanism) for sampling from the Laplace distribution for \( n = 6000 \) trials. A comparison of the outputs of the randomized rounding base-2 exponential mechanism with the \texttt{numpy} Laplace distribution for \( n = 6000 \) trials. (\( \varepsilon = 2 \ln(2), \gamma = 2^{-4}, O = \{\gamma i, -\gamma i \mid i \in [100]\}. \))
Figure 6: The maximum possible point-wise difference between the output probabilities of the unrounded naive mechanism and the randomized rounding exponential mechanism at varying granularities. Notice that as granularity increases, the error decreases.

Figure 7: Accuracy bounds for randomized rounding versus the probability assigned by the naive mechanism.
even when the floating point computations themselves are not revealed. To address these issues, we have specified an alternative view of DP using base 2 instead of base $e$ exponentiation. Base-2 DP has the benefit of allowing exact implementation of the exponential mechanism, without diverging from the logic of the original base-$e$ specification. By using exact computations, monitoring of correctness, which yields privacy, is more straightforward, and the mechanism is much harder to subvert as it is not vulnerable to attacks exploiting inexact arithmetic. We have also shown that exact implementation permits a rich set of privacy parameters and both integer and non-integer utility functions.

We have shown that the base-2 exponential mechanism is simple to implement and can take advantage of monitoring of exact arithmetic to ensure that computations are private and to alert the database administrator if any inexact and potentially privacy compromising arithmetic is performed. This implementation has no significant performance cost compared to a naive implementation for fixed utility ranges (e.g., generating Laplace noise) and utility ranges on the order of 20,000 can be executed successfully on consumer-level hardware. Furthermore, our handling of non-integer utility functions results in reasonable performance when simulating the Laplace distribution.

This work is a step towards a programming paradigm of exact arithmetic for DP. Future directions include more robust system architecture for monitoring and explicitly sequestering utility function execution, providing built-in utility methods, and extending the base-2 exponential mechanism to allow for custom weight computation logic in cases where naive weight computation is computationally infeasible (e.g., [2]) but precise computation of weights and avoiding zero-rounding are critical to privacy.

References

[1] V. Balcer and S. Vadhan. Differential privacy on finite computers. arXiv preprint arXiv:1709.05396, 2017.
[2] J. Blocki, A. Datta, and J. Bonneau. Differentially private password frequency lists. In NDSS, volume 16, page 153, 2016.
[3] C. Covington. Snapping Mechanism, 2019. URL https://github.com/ctcovington/floating_point.
[4] C. Dwork. Differential privacy. Encyclopedia of Cryptography and Security, pages 338–340, 2011.
[5] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. Foundations and Trends® in Theoretical Computer Science, 9(3–4):211–407, 2014.
[6] I. Gazeau, D. Miller, and C. Palamidessi. Preserving differential privacy under finite-precision semantics. Theoretical Computer Science, 655:92–108, 2016.
[7] T. Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library, 6.1.2 edition, 2019. URL http://gmplib.org/.
[8] A. Martelli and the gmpy2 contributors. General Multi-Precision arithmetic for Python, 2.1 edition, 2019. URL https://gmpy2.readthedocs.io/en/latest/.
[9] F. McSherry and K. Talwar. Mechanism design via differential privacy. In FOCS, volume 7, pages 94–103, 2007.
Given a privacy parameter \( \varepsilon \), a utility function \( u \) and an outcome set \( O \), the maximum difference in the probability of outputting an outcome \( o^* \in O \) between the randomized rounding exponential mechanism and the original exponential mechanism is upper-bounded by \( A + |B| \) where

\[
p_{u(d,o)} = |u(d,o) - [u(d,o)]|
\]

\[
q_{u,d,o} = \frac{p_{u(d,o)}e^{-\varepsilon[u(d,o)]}}{e^{-\varepsilon[u(d,o)]} + \mathbb{E}[\sum_{o' \in O \setminus \{o\}} e^{-\varepsilon\rho(u(d,o'))}]} + \frac{(1 - p_{u(d,o)})e^{-\varepsilon[u(d,o)]}}{e^{-\varepsilon[u(d,o)]} + \mathbb{E}[\sum_{o' \in O \setminus \{o\}} e^{-\varepsilon\rho(u(d,o'))}]}
\]

\[
A = 1 - \sum_{o \in O} q_{u,d,o}
\]

\[
B = \sum_{o \in O} e^{-\varepsilon u(d,o^*)} - q_{u,d,o^*}
\]

We break the proof of the lemma into several parts for easier reading. In part 1, we will show that \( q_{u,d,o} \) is indeed a lower bound for the probability that the randomized rounding exponential mechanism outputs \( o \). In part 2, we will show that \( A + |B| \) is an upper bound on the difference in probability that \( o^* \) is output by the randomized rounding exponential mechanism versus the original exponential mechanism.

**Part 1.** Given a constant \( a \) and a set of random variables \( X_i \), notice that \( \mathbb{E}[\frac{a}{a+\sum_i X_i}] \geq \frac{a}{a+\sum_i \mathbb{E}[X_i]} \).

This follows from observing that for a random variable \( Y \), \( \mathbb{E}[Y] \leq \frac{1}{\mathbb{E}[Y]} \) from Jensen’s inequality, and that \( \mathbb{E}[a + Y] = a + \mathbb{E}[Y] \) and \( \mathbb{E}[a + X_i] = a + \mathbb{E}[X_i] \) for independent random variables \( Y_1 \) and \( Y_2 \) by linearity of expectation.

The next step is to translate these observations to our scenario. Consider the expression for the expected value of the probability of selecting \( o \) in the randomized rounding exponential mechanism:

\[
\Pr[\mathcal{M}(u,d) = o] = \mathbb{E}[\frac{e^{-\varepsilon\rho(u(d,o))}}{\sum_{o \in O} e^{-\varepsilon\rho(u(d,o))}}]
\]

Notice that, as written, we cannot exactly translate the above using our previous observations as \( e^{\varepsilon\rho((u(d,o)))} \) is not a constant - it is a random variable. To get around this, we condition on \( u(d,o) \) being rounded either up or down:

\[
\Pr[\mathcal{M}(u,d) = o] = p_{u(d,o)} \Pr[\mathcal{M}(u,d) = o \text{ round down}] + (1 - p_{u(d,o)}) \Pr[\mathcal{M}(u,d) = o \text{ round up}]
\]

\[
= p_{u(d,o)} \mathbb{E}[\frac{e^{-\varepsilon[u(d,o)]}}{e^{-\varepsilon[u(d,o)]} + \sum_{o' \in O \setminus \{o\}} e^{-\varepsilon\rho(u(d,o'))}}] + (1 - p_{u(d,o)}) \mathbb{E}[\frac{e^{-\varepsilon[u(d,o)]}}{e^{-\varepsilon[u(d,o)]} + \sum_{o' \in O \setminus \{o\}} e^{-\varepsilon\rho(u(d,o'))}}]
\]

\[24\]
Now we have two terms with constant numerators, identical to 
\[ \frac{a}{a + \sum_i X_i} \] where
\[ X_i = e^{-\epsilon p(u(d,o_i))} \] and \[ a = e^{-\epsilon \left[p(u(d))\right]} \] or \[ e^{-\epsilon \left[q(u(d))\right]} \]. Thus,
\[ \Pr[M(u, d) = o] \geq \frac{p_u(d,o)e^{-\epsilon[u(d,o)]}}{e^{-\epsilon[u(d,o)]} + \sum_{o' \in O \setminus \{o\}} E[e^{-\epsilon p(u(d,o'))}]} + (1 - p_u(d,o)) e^{-\epsilon[u(d,o)]} + \frac{1}{e^{-\epsilon[u(d,o)]} + \sum_{o' \in O \setminus \{o\}} E[e^{-\epsilon p(u(d,o'))}]} \]
\[ \Pr[M(u, d) = o] \geq q_{u(d,o)} \]

Part 2. Call the probability that the randomized rounding mechanism outputs \( o \) \( s_o \). The difference in probability on any given element of the randomized rounding mechanism versus the original mechanism is \( s_o - \frac{e^{-\epsilon u(d,o)}}{\sum_{o \in O} e^{-\epsilon u(d,o)}} \). Taking the lower bound \( q_{u(d,o)} \) as an approximation for \( s_o \), notice that \( s_o - q_{u(d,o)} \leq 1 - \sum_{o \in O} q_{u(d,o)} \). This follows from observing that \( q_{u(d,o)} \) is a lower bound for \( s_o \) for all \( o \in O \), and thus the sum of the lower bounds is a lower bound on the sum, which is known to be 1. Thus the maximum error \( q_{u(d,o)} \) may have on any \( o \in O \) is \( A = 1 - \sum_{o \in O} q_{u(d,o)} \). Using \( q_{u(d,o)} \) as an approximation, the maximum error on \( \max\{A - B, B\} \leq A + |B| \).

\[ \Box \]

A.1 Sampling Procedure

Algorithm 3  The normalized weighted sampling algorithm without division

**Inputs:** \( W \), a set of weights.
**Outputs:** \( i \), an index sampled according to \( p(i) := \frac{w_i}{\sum_{w_j \in W} w_j} \).

1. **procedure** NORMALIZEDSAMPLE\((W)\)
2. \( t \leftarrow \sum_{w \in W} w \) \( \triangleright \) The total weight.
3. \( s \leftarrow \text{GetRandomValue}(p, t) \) \( \triangleright \) Sample a random value in \([0, t]\)
4. for \( i \in \{1, \ldots, |W|\} \) do
5. \( c_i \leftarrow \sum_{j=1} w_j \) \( \triangleright \) The cumulative weight for each element.
6. **return** \( \min\{i \mid c_i > s\} \) \( \triangleright \) Return the index associated with the region \( s \) landed in.

**Inputs:** the number of bits of precision \( p \), the upper bound of the range \([0, t]\) from which to sample.
**Output:** a number sampled uniformly at random from \([0, t]\).

7. **function** GETRANDOMVALUE\((p, t)\)
8. \( g \leftarrow \lfloor \log_2(t) \rfloor \) \( \triangleright \) The maximum power of 2 less than or equal to \( t \). We assume the floor operation rounds down.
9. Initialize \( s \leftarrow \infty \)
10. while \( s \geq t \) do \( \triangleright \) Reject \( s \) if it falls outside \([0, t]\)
11. for \( i \in \{1, \ldots, p\} \) do
12. \( r_i \sim \text{Unif}(0, 1) \) \( \triangleright \) Sample uniformly random bits.
13. \( s \leftarrow \sum_{i=1}^p r_i 2^{g-i} \) \( \triangleright \) Add up the powers of 2 indicated by \( r_i \) to produce a value in \([0, 2^{g-1}]\)
14. **return** \( s \)

The proof of correctness and the amount of randomness required for Algorithm 3 is identical to the logic for the proof of Lemma 3.9. The key difference is that this version always uses the same number of bits of randomness, regardless of the specific weights provided.
Proof of Corollary 3.4.1

Proof. The goal is to set integer $x, y$ and $z$ such that

$$\ln(2) \eta = \ln(2) z \log_2 \left( \frac{x}{2y} \right) = \varepsilon$$

Fixing $z = 1$,

$$\log_2 \left( \frac{x}{2y} \right) = \frac{\varepsilon}{\ln(2)}$$

Thus, $2^{\varepsilon/\ln(2)}$ can be approximated to within $(\frac{1}{2y})^{\ln(2)}$. The maximum error is therefore given by $t$ where

$$2^\varepsilon - t = 2^\varepsilon - 2^{-y \ln(2)}$$

Rearranging,

$$2^\varepsilon - t = 2^\varepsilon (1 - 2^{-y \ln(2) - \varepsilon})$$

Notice that as $y \to \infty$, $1 - 2^{-y \ln(2) - \varepsilon} \to 1$, and $t \to 0$. Thus, for arbitrarily large $y$, we can arbitrarily closely approximate $\varepsilon$. However, given a fixed maximum $y$ and a particular $\varepsilon$, we have that

$$-t = \log_2 (1 - 2^{-y \ln(2) - \varepsilon})$$

and thus for $y \geq -\ln(2)(\log_2(c) - \varepsilon)$, we can achieve error of less than $\log_2(1 - c)$.