Software and Systems Modeling
https://doi.org/10.1007/s10270-025-01298-x

SPECIAL SECTION PAPER

Quantifying Privacy Risk with Gaussian Mixtures

Rasmus C. Rønneberg[1] · Francesca Randone[2] · Raúl Pardo[3] · Andrzej Wąsowski[3]

Received: 10 May 2024 / Revised: 12 May 2025 / Accepted: 14 May 2025

Abstract
Data anonymization methods gain legal importance as data collection and analysis are expanding dramatically in data management and statistical research. Yet applying anonymization, or understanding how well a given analytics program hides sensitive information, is non-trivial. Privug is a method to quantify privacy risks of data analytics programs by analyzing their source code. The method uses probability distributions to model attacker knowledge and Bayesian inference to update said knowledge based on observable outputs. Currently, Privug is equipped with approximate Bayesian inference methods (such as Markov Chain Monte Carlo), and an exact Bayesian inference method based on multivariate Gaussian distributions. This paper introduces a privacy risk analysis engine based on Gaussian mixture models that combines exact and approximate inference. It extends the multivariate Gaussian engine by supporting exact inference in programs with continuous and discrete distributions as well as if-statements. Furthermore, the engine allows for approximating attacker knowledge that is not normally distributed. We evaluate the method by analyzing privacy risks in programs to release public statistics, differential privacy mechanisms, randomized response and attribute generalization. Finally, we show that our engine can be used to analyze programs involving thousands of sensitive records.

Keywords Privacy risk analysis · Bayesian inference · Probabilistic programming · Data analytics programs

1 Introduction

Data anonymization methods (also known as privacy protection mechanisms) gain legal importance [1] as data collection and analysis are expanding dramatically in data management and statistical research. Yet applying anonymization, or understanding how well a given analytics program hides sensitive information, is non-trivial [2]. Contemporary anonymization algorithms, such as differential privacy [3], require calibration to balance between reducing risks and preserving the utility of data. To assess the risks, data scientists need to assess the flow (leakage) of information from sensitive data fields to the output of analytics.

Measuring the information leakage is a useful technique to quantify how much an attacker may learn about the sensitive information a program processes. In the context of data anonymization programs, the goal is to determine whether the sensitive input data does not leak into the output of the anonymization program—as otherwise the data anonymization program has failed to accomplish its task. Information leakage is traditionally measured using probability and information-theoretic measures [4]. For instance, it is possible to answer queries such as *"If we release the (possibly anonymized) average income of the employees in a company, what is the probability of the attacker guessing the income of an individual employee correctly?"* or *"How much information (measured in terms of entropy) is shared between the income of individual employees and the average income?"*—intuitively these queries are instances of two information leakage queries, namely, the Bayes vulnerability and the mutual information query.

Communicated by Carla Ferreira and Tim Willemse.

✉ Rasmus C. Rønneberg
rasmus.ronneberg@kit.edu

Francesca Randone
francesca.randone@imtlucca.it

Raúl Pardo
raup@itu.dk

Andrzej Wąsowski
wasowski@itu.dk

1 Karlsruhe Institute of Technology, Karlsruhe, Germany

2 University of Trieste, Trieste, Italy

3 IT University of Copenhagen, Copenhagen, Denmark

Many methods have been proposed to quantify information leakage [4–12]. Some methods use techniques to approximate information leakage measures [5–8, 10–12] whereas others focus on exact reasoning [9, 13]. Methods based on approximations produce less accurate results than exact methods; which may lead to missing data leaks. However, exact methods often feature low scalability or cannot analyze complex programs. These trade-offs impact the applicability of different methods to analyze data anonymization programs in realistic data analytics systems.

Privug is a tool-supported method for privacy risk analysis (via information leakage) that has been used to analyze modern anonymization methods and data analytics systems [12–15]. It relies on Bayesian inference to quantify privacy risks in data analytics programs. The attacker's knowledge is modeled as a probability distribution over program inputs, and it is then conditioned on the disclosed program outputs. Then, Bayesian probabilistic programming is used to compute the posterior attacker knowledge, i.e., the updated attacker knowledge after observing the outputs in the program. Using prior and posterior attacker knowledge, it is possible to compute most information leakage measures [4]. In previous works, we have used Privug to compute metrics in the g-vulnerability family such as Bayes vulnerability [14] and entropy-based measures such as Shannon entropy, KL-divergence, and mutual information [12, 13]. Furthermore Privug works directly on the program source code, which allows for directly analyzing the code used to process the data. The Privug method can be used to analyze Python code via the Privugger library. As of today, it is equipped with approximate Bayesian inference methods [12] (Markov Chain Monte Carlo [16]) and an exact Bayesian inference method [13]. The former can be used to analyze arbitrary Python programs. However, results may be inaccurate and it is computationally expensive for some programs. The latter requires that attacker's knowledge is modeled using a multivariate Gaussian distribution, and only supports programs that perform closed-form operations on this type of distribution—this excludes programs with if-statements or discrete distributions. But it produces exact results and it is highly scalable. In the domain of privacy risk analysis using Bayesian inference, the trade-off between exact and approximate analysis, program expressiveness and scalability remains unexplored.

We present a novel privacy risk analysis engine for Privug that combines *exact* and *approximate* Bayesian inference. The method uses Gaussian mixture models to capture attacker's knowledge. This work constitutes a step forward with respect to our earlier exact inference engine [13] (cf. Sect. 5). The engine presented in this paper can perform exact inference on all programs supported in [13]. Furthermore, using Gaussian mixtures, we can perform exact inference on programs that combine continuous and discrete distributions, and also if-statements. This enables the possibility of analyzing privacy protection mechanisms such as randomized response (cf. 4.2) or attribute generalization (cf. 4.3), which were not supported in [13]. Finally, Gaussian mixtures can be used for approximating arbitrary distributions [17], which allows for approximating attacker knowledge that is not normally distributed. This increases the flexibility in modeling attacker's prior knowledge. For instance, it allows us to to define discrete distributions (e.g., if we are modeling attacker knowledge about a count variable), restrict uncertainty to a closed interval (as opposed to using Gaussian distributions whose support is $\pm\infty$), or use distributions flatter than Gaussian (e.g., uniform) to model uncertainty in prior attacker knowledge.

This work constitutes a new point in understanding the trade-off among exact and approximate inference, program expressiveness and performance in quantification of privacy risks by means of Bayesian inference. Specifically, our contributions are:

1. A probabilistic programming language to perform privacy risk analyses that supports exact and approximate Bayesian inference. The language is a subset of Python.
2. An operational semantics for the language based on Gaussian mixture models.
3. A prototype implementation of the privacy risk analysis method.
4. An evaluation of the privacy risk analysis engine in multiple case studies: release of public statistics, three differential privacy mechanisms (Gaussian and Laplace [3], and randomized response [18]), and attribute generalization [19].
5. A scalability evaluation showing that our engine can analyze large systems involving thousands of individuals. The evaluation compares the engine with our previous work [13].

2 Background

2.1 Privug: A Data Privacy Debugging Method

Let $\mathbb{I}$, $\mathbb{O}$ denote sets of *inputs* and *outputs*, respectively. We use $\mathcal{D}(\mathbb{I})$ to denote a space of distributions; in this case over inputs. Let $d \in \mathcal{D}(\mathbb{I})$ denote a distribution over inputs, $I \sim \mathcal{D}(\mathbb{I})$ denotes a random variable distributed according to d. Privug is a method to explore information leakage on data analytics programs [12]. It combines a probabilistic model of attacker knowledge with the program under analysis to quantify privacy risks. The process takes the following steps:

(1) Prior We first model the *prior* knowledge of an attacker as a distribution over program inputs $\mathbb{I}$. This distribution represents the input values of a program that the attacker

finds plausible. For example, consider a program that takes as input a real number A representing the age of an individual ($\mathbb{I} \triangleq \mathbb{R}$). A possible prior knowledge of the attacker could be: $A \sim \mathcal{U}(0, 120)$ (all ages between 0 and 120 are equally likely from what the attacker knows) or $A \sim \mathcal{N}(\mu = 42, \sigma = 2)$ (the attacker believes that the age of 42 and values nearby are most likely ages). We write $\Pr(I)$ for the distribution of prior attacker knowledge.

(2) Probabilistic program interpretation The second step is to interpret a target program $\pi : \mathbb{I} \to \mathbb{O}$ using the attacker prior knowledge. To this end, we lift the program to run on distributions $\mathcal{D}(\mathbb{I})$ instead of concrete inputs $\mathbb{I}$. This corresponds to the standard lifting to the probability monad [20]; *lift* : $(\mathbb{I} \to \mathbb{O}) \to (\mathcal{D}(\mathbb{I}) \to \mathcal{D}(\mathbb{O}))$. For example, consider the following program that computes the average age of a list of ages (in Python):

Listing 1 A program computing a descriptive statistics (mean) for a list of ages

```python
def average_age(ages: List[float]):
    return sum(ages)/len(ages)
```

The lifted version of the program is:

Listing 2 The program of Lst. 1 lifted to beliefs (distributions) about secret inputs

```python
def average_age(ages: Dist[List[float]]):
    return sum(ages)/len(ages)
```

where `Dist[List[float]]` denotes a distribution over lists of floats, $\mathcal{D}(\mathbb{R}^n)$. Python allows retaining the same body, thanks to automatic vectorization of operators over distributions. In most statically programming languages, and especially in functional programming languages, one would explicitly use the `map` operator over distributions. The lifted program yields the distribution $\Pr(O \mid A)$, so the probability of (the belief in) each program output O, given the knowledge of the attacker about the input age A. In general, the combination of prior attacker knowledge with the lifted program yields a joint distribution on inputs and outputs: $\Pr(O \mid I) \Pr(I) = \Pr(O, I)$.

(3) Observations One key use case of Privug is to understand what the attacker can infer about the secret input, once a concrete output of the program is released. To this end, one may add observations to the probabilistic model. In the average example in Listing 2, we could check how the knowledge of the attacker changes when the attacker observes that the average is 44. This step yields the posterior distribution $\Pr(I \mid O = 44)$. In general, a likelihood function on the joint distribution of input and output (defined by the probabilistic lifting of the program π), $\Pr(E \mid I, O)$, can be used to observe a predicate E on the joint distribution. The observation of predicate E can be exact (the attacker observes precisely the predicate E), or noisy (the attacker observes an approximation of predicate E, i.e., partial observability).

(4) Posterior Inference The next step is to apply Bayesian inference to obtain a posterior distribution on the input variables. This allows to understand the attacker's knowledge about the input (and the actual output) given the observation E.

$$\Pr(I, O \mid E) = \frac{\Pr(E \mid I, O) \Pr(O \mid I) \Pr(I)}{\Pr(E)} \tag{1}$$

The term $\Pr(E)$ is a summation ($\sum_{O,I} \Pr(E|I, O) \Pr(O|I) \Pr(I)$), integral ($\int_{O,I} \Pr(E|I, O) \Pr(O|I) \Pr(I) \mathrm{d}O\mathrm{d}I$), or combinations of both over the domain of input/output variables. These domains are often very large (or even infinite), and can also be uncountable (in the case of continuous distributions). As a consequence, it is usually intractable to compute a symbolic representation of $\Pr(E)$. Therefore, it is not possible to get analytical solutions for the posterior distributions. Existing implementations of Privug use Markov Chain Monte Carlo (MCMC) [16] or affine transformations of multivariate Gaussian distributions to tackle this issue. But these methods are either approximate or limited in terms of the programs they support. As mentioned above, the subject of this paper is to explore the use of an inference method based on Gaussian mixtures that combines exact and approximate inference.

(5) Posterior analysis We query the posterior and prior distributions (attacker knowledge) to measure how much the attacker has learned. Different techniques and metrics can be applied to this end. Information-theoretic metrics such as entropy and mutual information (e.g., [21]) can be used to model attacker uncertainty about the secret and correlation between the secret and public outputs, respectively. For instance, low entropy values for attacker posterior knowledge about a secret indicates that the attacker is certain about the value of the secret. Similarly, a value close to zero of mutual information between a secret input and public output (in the attacker posterior knowledge) indicates a high correlation between the output and the input. Since the attacker has access to the public output, low mutual information is indicative of information leakage. Quantitative information flow metrics such as Bayes vulnerability focus on measuring the probability of attackers guessing the secret [4]. For instance, Bayes vulnerability measures the expected probability of an attacker guessing the secret in one try, after observing the output of the program. The g-vulnerability family of metrics allows modeling a broad range of attackers [4]. Finally, it is also possible to query and plot visualizations of attacker prior/posterior knowledge to characterize what information about the secret the attacker learned. This is the type of analysis we use in this paper, although the Privug method can also be used to compute the aforementioned metrics [12, 14]. We illustrate the use of probability queries and visualizations of attacker knowledge for the average age program above. Figure 1 compares the prior and posterior distribu-

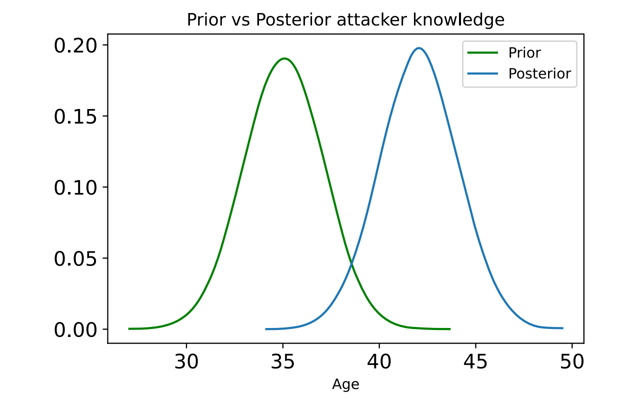


Fig. 1 Prior (Left) vs Posterior (Right) attacker knowledge on ages for average age program above

tions of attacker knowledge. We analyze the case where the output of the program is 44. The green line shows the prior attacker knowledge on the victim's age $\Pr(A)$ and the blue line the posterior knowledge $\Pr(A \mid O = 44)$ when observing that the program output is 44. The prior attacker knowledge is $A \sim \mathcal{N}(35, 2)$ for the victim's age and also for the rest. The figure clearly shows that the attacker now believes that higher ages are more plausible. In other words, the attacker prior knowledge has been corrected towards more accurate knowledge on the victim's age.

2.2 Multivariate Gaussian Distributions

In this paper, we use capital Greek letters for matrices, and bold font for column vectors. Small letters a and b are reserved for selecting subvectors (as in $\boldsymbol{\mu}_a$) and pairs of them for selecting submatrices (as in Σ_{ba}). Matrix and vector literals are written in brackets. We write $supp(X)$ for the support of the random variable X.

A *multivariate Gaussian distribution*, denoted $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, defines a probabilistic model composed of n normally distributed random variables, $\boldsymbol{X} = [X_1, X_2, \dots, X_n]^\intercal$. The distribution is parameterized by a vector $\boldsymbol{\mu}$ of n means, and a symmetric $n \times n$ *covariance* matrix Σ, so $\Sigma_{ij} = \mathrm{cov}[X_i, X_j]$ gives the covariance between variables X_i and X_j, while Σ_{kk} gives the variance of variable X_k. To ensure that the (degenerate) Gaussian distributions are well-defined, we assume that the covariance matrix is semi-positive definite. The probability density function for this multivariate Gaussian is:

$$\phi(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma) = \\ ((2\pi)^n |\Sigma|)^{-1/2} \exp\left(-2^{-1}(\boldsymbol{x} - \boldsymbol{\mu})^\intercal \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu})\right) \tag{2}$$

where $|\Sigma|$ denotes the determinant of the matrix Σ. We use $\Phi(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$ to denote the corresponding cumulative density function of multivariate Gaussian distributions.

We recall standard properties of multivariate Gaussian distributions [22–24].

Theorem 1 *Let* $\begin{bmatrix} \boldsymbol{X}_a \\ \boldsymbol{X}_b \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ *with* $\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{bmatrix}$ *and* $\Sigma = \begin{bmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{bmatrix}$.
The marginal distributions are $\boldsymbol{X}_a \sim \mathcal{N}(\boldsymbol{\mu}_a, \Sigma_{aa})$, $\boldsymbol{X}_b \sim \mathcal{N}(\boldsymbol{\mu}_b, \Sigma_{bb})$, *and* $\boldsymbol{X}_i \sim \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_{ii})$ *for* $i = 1 \dots a + b$.

The covariance matrix identifies independent random variables:

Theorem 2 *Let* $[X_1, \dots, X_n]^\intercal \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, *two marginals* X_i, X_j *with* $i \neq j$ *are* independent *iff* $\Sigma_{ij} = cov[X_i, X_j] = 0$.

The space of Gaussian distributions is closed under affine transformations:

Theorem 3 *Let* $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ *and* $\boldsymbol{Y} = \mathrm{A}\boldsymbol{X} + \boldsymbol{b}$ *be an affine transformation with* $\mathrm{A} \in \mathbb{R}^{m\times n}$ *a projection matrix and* $\boldsymbol{b} \in \mathbb{R}^{n\times 1}$ *a column vector. Then we have that:*

$$\boldsymbol{Y} \sim \mathcal{N}(\mathrm{A}\boldsymbol{\mu} + \boldsymbol{b}, \mathrm{A}\Sigma\mathrm{A}^\intercal) \ .$$

We use $Y \mid X_1, X_2, \dots, X_n$ to denote a random variable Y that is distributed conditionally with respect to $X_1, X_2, \dots, X_n$. Linear combinations of random variables can be used to define hierarchical probabilistic models consisting of dependent random variables, such as Gaussian Bayesian networks [23].

Theorem 4 *Let* $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ *and* $Y \mid \boldsymbol{X} \sim \mathcal{N}(\boldsymbol{a}^\intercal \boldsymbol{X} + b, \sigma^2)$, *where* $\boldsymbol{a} \in \mathbb{R}^{n\times 1}$ *is a vector,* $b \in \mathbb{R}$ *and* $\sigma^2 > 0$. *Then* $[\boldsymbol{X}^\intercal, Y]^\intercal \sim \mathcal{N}([\boldsymbol{\mu}^\intercal, \boldsymbol{a}^\intercal\boldsymbol{\mu} + b]^\intercal, \Sigma')$ *with*

$$\Sigma'_{1..n,1..n} = \Sigma, \ \Sigma'_{(n+1)(n+1)} = \sigma^2 + \boldsymbol{a}^\intercal \Sigma \boldsymbol{a}, \ \Sigma'_{i(n+1)} \\ = cov[X_i, Y] = \textstyle\sum_{j=1}^{n} a_j \Sigma_{ij} \ .$$

Example 1 We present an example of a Gaussian Bayesian network [23]. Let $X_1 \sim \mathcal{N}(50, 2)$, $X_2 \mid X_1 \sim \mathcal{N}(2X_1 - 5, 1)$, and $X_3 \mid X_2 \sim \mathcal{N}(X_2 - 10, 4)$. Here the distribution of X_2 is conditioned on X_1, and of X_3 on X_2. The probabilistic model defines a joint multivariate Gaussian probability distribution $[X_1, X_2, X_3]^\intercal \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$. Theorem 4 allows to compute the mean, variance, and covariance of this joint distribution:

$$\boldsymbol{\mu} = \begin{bmatrix} 50 \\ 2 \cdot \boldsymbol{\mu}_1 - 5 \\ 1 \cdot \boldsymbol{\mu}_2 - 10 \end{bmatrix} = \begin{bmatrix} 50 \\ 95 \\ 85 \end{bmatrix},$$

$$\Sigma = \begin{bmatrix} 2 & 2 \cdot \Sigma_{11} & 0 + 1 \cdot \Sigma_{12} \\ & 1 + 2 \cdot \Sigma_{11} \cdot 2 & 0 + 1 \cdot \Sigma_{22} \\ & & 4 + 1 \cdot \Sigma_{22} \cdot 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 4 \\ & 9 & 9 \\ & & 13 \end{bmatrix}$$

As the matrices are symmetric, we only show the upper-right triangle. Note that, even though X_3 does not directly depend on X_1, it still has a non-zero covariance. The reason for this is the indirect dependence through X_2. □

We use *conditioning* to model observations on values of random variables.

Theorem 5 *Let $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ be split into two sub-vectors so that*

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_a \\ \boldsymbol{X}_b \end{bmatrix}, \quad \boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{bmatrix},$$
$$\Sigma = \begin{bmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{bmatrix} \text{ and } \boldsymbol{x}_b \in supp(\boldsymbol{X}_b).$$

The conditioned distribution is $\boldsymbol{X}_a \mid (\boldsymbol{X}_b = \boldsymbol{x}_b) \sim \mathcal{N}(\boldsymbol{\mu}', \Sigma')$ with $\boldsymbol{\mu}' = \boldsymbol{\mu}_a + \Sigma_{ab}\Sigma_{bb}^{-}(\boldsymbol{x}_b - \boldsymbol{\mu}_b)$ and $\Sigma' = \Sigma_{aa} - \Sigma_{ab}\Sigma_{bb}^{-}\Sigma_{ba}$, where Σ^{-} is the generalized inverse.

Example 2 Consider the multivariate distribution of Example 1. We condition X_3 to be 85. By Thm. 5 the posterior of $X_1, X_2 | X_3 = 85$ is $\mathcal{N}(\boldsymbol{\mu}', \Sigma')$ with

$$\boldsymbol{\mu}' = \begin{bmatrix} 50 \\ 95 \end{bmatrix} + \begin{bmatrix} 4 \\ 9 \end{bmatrix} \begin{bmatrix} 13 \end{bmatrix}^{-} (85 - 85) = \begin{bmatrix} 50 \\ 95 \end{bmatrix}$$
$$\text{and } \Sigma' = \begin{bmatrix} 2 & 4 \\ 4 & 9 \end{bmatrix} - \begin{bmatrix} 4 \\ 9 \end{bmatrix} \begin{bmatrix} 13 \end{bmatrix}^{-} \begin{bmatrix} 4 & 9 \end{bmatrix} = \begin{bmatrix} 10/13 & 16/13 \\ 16/13 & 36/13 \end{bmatrix}$$

□

2.3 Gaussian Mixtures

Mixtures are the scalar products of two vectors $(\pi_1, \ldots, \pi_C)$ and $(D_1, \ldots, D_C)$ such that $\sum_{i=1}^{C} \pi_i = 1$, $0 < \pi_i \leq 1$, and D_i is the distribution of the i-th *component*, with $i = 1, \ldots, C$. The numbers π_i are called *weights*. We denote a mixture as $M = \pi_1 D_1 + \ldots + \pi_C D_C$, thus indicating that M has probability density function $f_M(x) = \pi_1 f_{D_1}(x) + \ldots + \pi_C f_{D_C}(x)$ where f_{D_i} are the probability density functions of the components. When $C = 1$, we recover the case of a single distribution. Intuitively a mixture can be seen as a particular distribution in which the samples come from component D_i with probability π_i.

A special case of mixture distributions is given by Gaussian Mixtures (GMs) in which every component is a multivariate Gaussian, i.e. $D_i = \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$ for $i = 1, \ldots, C$, with mean vectors and covariance matrices $\boldsymbol{\mu}_i, \Sigma_i$. We assume $(\boldsymbol{\mu}_i, \Sigma_i) \neq (\boldsymbol{\mu}_j, \Sigma_j)$ for $i \neq j$, since identical components could be merged into a single one. The set of GMs is dense in the set of probability distributions with respect to the weak topology [25], meaning that for any probability distribution one can always find a GM that approximates it arbitrarily closely with respect to the Levy-Prokhorov distance [26]. Since Dirac deltas can be seen as particular Gaussian distributions, having null covariance matrix, discrete distributions over a finite set of values, i.e. mixtures of deltas, are included in the set of GMs.

Since the probability density function of a GM is the weighted sum of Gaussian densities, all the theorems stated above for Gaussian distributions, can be lifted by linearity to GMs. We show how this applies to marginalization and affine transformation in the following example.

Example 3 We show how the theorems of the previous section can be lifted to GMs by considering the special case of a two-component mixture. For a larger number of components the procedure is analogous. Consider

$$\begin{bmatrix} \boldsymbol{X}_a \\ \boldsymbol{X}_b \end{bmatrix} \sim \pi_1 \mathcal{N}(\boldsymbol{\mu}_1, \Sigma_1) + \pi_2 \mathcal{N}(\boldsymbol{\mu}_2, \Sigma_2)$$

where $\boldsymbol{\mu}_i = \begin{bmatrix} (\boldsymbol{\mu}_i)_a \\ (\boldsymbol{\mu}_i)_b \end{bmatrix}$ and $\Sigma_i = \begin{bmatrix} (\Sigma_i)_{aa} & (\Sigma_i)_{ab} \\ (\Sigma_i)_{ba} & (\Sigma_i)_{bb} \end{bmatrix}$ for $i = 1, 2$. Then, computing the marginal with respect to $\boldsymbol{X}_a$ corresponds to integrating the probability density function of the Gaussian mixture with respect to x_b, i.e. computing

$$\int_{-\infty}^{+\infty} \left(\pi_1 \phi(x \mid \boldsymbol{\mu}_1, \Sigma_1) + \pi_2 \phi(x \mid \boldsymbol{\mu}_2, \Sigma_2)\right) dx_b.$$

By linearity of the integral this is equivalent to computing

$$\pi_1 \int_{-\infty}^{+\infty} \phi(x \mid \boldsymbol{\mu}_1, \Sigma_1) dx_b + \pi_2 \int_{-\infty}^{+\infty} \phi(x \mid \boldsymbol{\mu}_2, \Sigma_2) dx_b$$

which is exactly the weighted sum of the components' marginals with respect to $\boldsymbol{X}_a$, computable using Theorem 1. Therefore the marginal of the GM is exactly $\pi_1 \mathcal{N}((\boldsymbol{\mu}_1)_a, (\Sigma_1)_{aa}) + \pi_2 \mathcal{N}((\boldsymbol{\mu}_2)_a, (\Sigma_2)_{aa})$.

Now let $\boldsymbol{X} \sim \pi_1 \mathcal{N}(\boldsymbol{\mu}_1, \Sigma_1) + \pi_2 \mathcal{N}(\boldsymbol{\mu}_2, \Sigma_2)$ and $\boldsymbol{Y} = \mathrm{A}\boldsymbol{X} + \boldsymbol{b}$. The density of $\boldsymbol{Y}$ can be computed applying a change of variable to the density of $\boldsymbol{X}$ but, by linearity, this amounts to performing a change of variable in the density of each component, computable using Theorem 3. Therefore, $\boldsymbol{Y} \sim \pi \mathcal{N}(\mathrm{A}\boldsymbol{\mu}_1 + \boldsymbol{b}, \mathrm{A}\Sigma_1\mathrm{A}^\intercal) + (1 - \pi)\mathcal{N}(\mathrm{A}\boldsymbol{\mu}_2 + \boldsymbol{b}, \mathrm{A}\Sigma_2\mathrm{A}^\intercal)$.

3 Gaussian Mixture Engine for Privug

Our inference engine is an interpreter of a probabilistic programming language that captures a subset of Python. Probabilistic programming languages combine random variables, deterministic variables, standard program statements and conditioning (see, e.g., [27]). Our target probabilistic programming language includes variable assignments, bounded for-loops, if-statements, binary operators, sequencing, *probabilistic assignments*, and *observations* (conditioning). Let $v_r \in \mathbb{R}$ be real values, $x, y, z, \ldots$ denote Dirac delta random variables (used for representing deterministic variables and discrete distributions), $X, Y, Z, \ldots$ denote (Gaussian distributed) random variables, and $\boldsymbol{X}$ a vector of random variables. Let $\oplus \in \{+, -, *, /\}$, and $\odot \in \{<, >, \leq, \geq, =, \neq\}$. The syntax of well-formed programs is generated by the rule p below.

$$\begin{array}{rrcl}
\text{(Expressions)} & e & ::= & v_r \mid x \mid e \oplus e \\
\text{(Distributions)} & d & ::= & \texttt{Normal}(e,\ e) \mid \\
 & & & \texttt{Normal}(e * X + e,\ e) \mid \\
 & & & \texttt{D}(e, \ldots, e) \\
\text{(Conditions – } \texttt{if}\text{)} & \varphi & ::= & x \odot e \mid X \leq e \mid X \geq e \\
\text{(Conditions – } \texttt{cond}\text{)} & \psi & ::= & \varphi \mid X = e \\
\text{(Statements)} & s & ::= & X = d \mid X = Y \oplus e \mid \\
 & & & X = Y + Z \mid \\
 & & & \texttt{condition}(\psi) \mid \\
 & & & x = e \mid x = \texttt{D}(e, \ldots, e) \mid \\
 & & & s\texttt{;}\ \ s \mid \texttt{if}\ \ \varphi \\
 & & & \texttt{:}\ s\ \ \ \texttt{else:}\ \ s \mid \\
 & & & \texttt{for}\ x\ \texttt{in range}\ v_r\ s \\
\text{(Programs)} & p & ::= & s\texttt{;}\ \ \texttt{return}\ \boldsymbol{X}
\end{array}$$

We admit (e) constants expressions, references to Dirac delta random variables, and binary operations. Two ways of defining normal distributions (d) are supported: an independent Gaussian distribution, or a linear transformation of Gaussian random variables. The syntax also allows for specifying custom distributions. We admit Boolean conditions modeling (in)equalities over Dirac delta variables and expressions. For (Gaussian distributed) random variables, we admit simple inequalities in if-statements, and, additionally, equality for condition statements. We rule out equality tests for these variables as they are probability 0 events without closed form solution for if-statements. This corresponds to discrete and continuous distributions encoded as Gaussian mixtures (see Sect. 3.1 for details). Statements (s) are: Gaussian probabilistic assignments (a normal, a custom distribution, a transformed random variable, a sum of two random variables), an observation (conditioning), Dirac delta assignments (constants and discrete distributions), sequencing, if-statements and a limited for-loop. We define no expressions over Gaussian random variables, only statements, to simplify introduction of changes to the state (the probabilistic model) in the semantics for each sub-expression (Sect. 3.1). The for-loops are only a convenience construct for repetitive statements. A program (p) terminates returning a sub-vector of random variables (`return`). The distribution of the returned variable is the marginal of the posterior joint probability distribution to analyze for privacy risks.

We show in Sect. 4 that this syntax captures realistic case studies involving different types of data anonymization methods.

Concretely, we evaluate three widely-used differential privacy mechanisms (randomized response, the Gaussian mechanism and the Laplace mechanism) and attribute generalization (which is one of the anonymization steps in the popular k-anonymity algorithm [28]).

3.1 Gaussian Mixture Semantics

The formal semantics is defined in the small-step style, over terms of Gaussian mixtures (Sect. 2.2). It provides an engine that combines exact and approximate inference to track attacker knowledge in Privug (cf. Sect. 2.1).

A *state* $\mathcal{S}$ is a Gaussian mixture $\sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$ modeling the joint probability distribution of all program variables. We use $\mathcal{G}$ to denote single multivariate Gaussian distributions. The multivariate Gaussian of each mixture component defines the mean and covariance of n random variables. Deterministic program variables are modeled as Dirac delta distributions; i.e., they have a fixed mean, and (co)variance 0. We use $\boldsymbol{\mu}_X$ to denote the mean of marginal variable X in the mean vector $\boldsymbol{\mu}$. Given a covariance matrix Σ we use $\Sigma_{(X,X)}$ to denote the variance of the marginal variable X in Σ, and $\Sigma_{(.,X)}$, $\Sigma_{(X,.)}$ to denote the covariance vectors of X with the other variables in the multivariate Gaussian. Similarly, given a sub-vector of random variables $\boldsymbol{X}$, we use $\boldsymbol{\mu}_{\boldsymbol{X}}$ and $\Sigma_{\boldsymbol{X}}$ to denote the mean vector and covariance of the variables in the sub-vector $\boldsymbol{X}$. For clarity, when referring to variables in the mean vector or covariance matrix of a Gaussian mixture component we use parenthesis—e.g., $(\boldsymbol{\mu}_i)_X$ or $(\Sigma_i)_{(X,X)}$—to separate the indexes of Gaussian mixture components and variables. We use the syntax $\boldsymbol{\mu}' = \boldsymbol{\mu}[X = c]$ to denote a vector $\boldsymbol{\mu}'$ whose X component equals c and the rest remain as defined in $\boldsymbol{\mu}$—and analogously for Σ. If a marginal variable is a Dirac delta, then we use lower case x to index it. We consider an initial state composed by a single component Gaussian mixture $1 \cdot \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ where the mean vector ($\boldsymbol{\mu}$) and covariance matrix (Σ) include entries for all program variables and are initialized to 0. Consequently, we require that well-formed programs do not read unassigned variables; as otherwise they might incorrectly read 0 Dirac delta distributions for uninitialized variables. We remark that this constraint is used only to simplify the presentation of the semantics. In practice, we may introduce variables into the Gaussian mixture components as they are assigned in the program.

Definition 1 (Semantics) The semantics is given by the relations $\rightarrow_e\colon e \times \mathcal{G} \rightarrow \mathbb{R}$ and $\rightarrow_s\colon s \times \mathcal{S} \rightarrow \mathcal{S}$ for expressions

Fig. 2 Term evaluation rules ($\rightarrow_e$). $\mathcal{G}$ stands for $\mathcal{N}(\boldsymbol{\mu}, \Sigma)$

$$(\textbf{V-Exp})\ \frac{\boldsymbol{\mu}_x = c \qquad \Sigma_{(x,x)} = 0 \qquad \Sigma_{(x,.)} = \mathbf{0}^{\mathrm{T}} \qquad \Sigma_{(.,x)} = \mathbf{0}}{\langle x, \mathcal{G}\rangle \rightarrow_e c}$$

$$(\textbf{O-Exp})\ \frac{\langle e_0, \mathcal{G}\rangle \rightarrow_e c_0 \qquad \langle e_1, \mathcal{G}\rangle \rightarrow_e c_1}{\langle e_0 \oplus e_1, \mathcal{G}\rangle \rightarrow_e c_0 \oplus c_1} \qquad (\textbf{C-Exp})\ \frac{}{\langle c, \mathcal{G}\rangle \rightarrow_e c}$$

e and statements s, respectively, as defined in Fig. 2 (expressions), Figs. 3 and 4 (statements).

The rules in Fig. 2 operate on a single multivariate Gaussian; as expressions are evaluated for each component separately. The rules in Fig. 3 include non-conditional program statements: non-conditional control flow statements (as the for-loop is syntactic sugar for repeated sequences of statements), assignments, and arithmetic operations over Gaussian random variables. The rules in Fig. 4 define conditional program statements: conditioning and if-statements. In what follows we explain the details of the rules.

The rules for the evaluation expressions (Fig. 2) are standard. The only specific characteristic is the evaluation of deterministic variables (V- Exp) , i.e. random variables whose probability mass is concentrated in a single point. Since they must be modeled as Dirac delta distributions, we check that the (co)variance is 0 and return the value specified in the mean vector. Note that expressions are evaluated on a single multivariate Gaussian, this is because the different components of the Gaussian mixture produce different expression evaluations.

The rules for sequence of statements (Seq) and for-loops (For- B, For- I) are standard, and they do not change the state's Gaussian mixture. We omit their details. We remark that, in the first iteration of the (For- I) rule, variable $x = 0$ as per initialization of the state—we consider this implicit initialization part of well-formed programs to keep the presentation of the rule as simple as possible. Programs finish with a `return` $\boldsymbol{X}$ instruction. It returns a Gaussian mixture $\sum_i \pi_i \cdot \mathcal{N}((\boldsymbol{\mu}_i)_{\boldsymbol{X}}, (\Sigma_i)_{\boldsymbol{X}})$ with the marginal variables specified sub-vector ($\boldsymbol{X}$) of the state's Gaussian mixture (Ret). In what follows, we focus on the rules manipulating the state's Gaussian mixture.

There are four types of assignments: deterministic assignments, assignments of independent and linearly dependent Gaussian distributions, and assignments of custom distributions. All types of assignments require updating all components of the Gaussian mixture. Deterministic assignment (D- Asg) update the mean of the specified variable $(\boldsymbol{\mu}_i)_x$ with the result of evaluating expression e (in each component). The covariance matrix needs not to be updated as it is initialized to 0, and deterministic variables are modeled as Dirac delta distributions. Independent assignments of Gaussian distributions (P- Asg- Ind) also set the mean of X in the mean vector. The variance X is set by updating its entry in the covariance matrix $(\Sigma_i)_{(X,X)}$; as before we do not need to update covariances since they were initialized to 0. Dependent assignments of Gaussian distributions (P- Asg- Dep) update to the mean vector with a mean computed as a linear combination with the mean of the dependent random variable Y. The variance of X is set in $(\Sigma_i)_{(X,X)}$. Note that the variance is a linear combination with the variance of Y. In this case, we also set the covariances of X using the covariance vectors of Y. This is because the new variable depends on Y and consequently on all the variables that Y depends on. Let $\mathcal{D}$ denote the domain of univariate distributions. The assignment of custom distributions (P- Cus- Asg) uses the operator $\texttt{match}_2 : \mathcal{D} \rightarrow \mathcal{S}$ which takes as input an arbitrary distribution $D \in \mathcal{D}$ and returns a Gaussian mixture G satisfying the following properties: i) the operator is finitely computable for any distribution D, ii) G has the same mean and variance as D. Observe that one can always define $\texttt{match}_2$ taking G to be the Gaussian distribution with mean and variance equal to that of D, therefore the existence of at least one operator satisfying i) and ii) is guaranteed. At the moment we do not give an operational definition of $\texttt{match}_2$, and define the semantics for any operator satisfying these two properties. In particular, we require that i) $\texttt{match}_2(D)$ is computable and ii) that the first two moments are exactly preserved in the Gaussian mixture approximation. This aligns with the more general construction of the moment-matching operator presented by Randone and coauthors [17]. Example 4 illustrates how to define $\texttt{match}_2$ for Bernoulli distributions. This definition can be generalized to any discrete distribution. Bernoulli distributions can be seen as a special case of discrete distributions having support in $\{0, 1\}$ and probability masses $\{p, 1 - p\}$. For general discrete distributions D with support in $\{n_1, \ldots, n_d\}$ and probability masses $\{p_1, \ldots, p_d\}$ we can set $\texttt{match}_2(D) = \sum_{i=1}^{d} p_d \mathcal{N}(n_d, 0)$. In Section 4.3, we give a definition of $\texttt{match}_2$ for uniform and Laplacian distributions that maps the distributions to Gaussian mixtures of more than one component, but we remark that different definitions can be given as long as i) and ii) hold. When $\texttt{D}(e, \ldots, e)$ is a discrete distribution, the rule (P- Cus- Asg) corresponds to the statement $x = \texttt{D}(e, \ldots, e)$; we omit it in Fig. 3 for simplicity.

Example 4 Consider the program $X = \texttt{Normal}(15, 2)$; $x = 20$; $Z = \texttt{Normal}(2X, 1)$. The first and second assignments updates the state, according to P- Asg- Ind and D- Asg, respectively, into

Fig. 3 Operational Semantics rules for non-conditional program statements; $\mathcal{S}$ stands for $\sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$

$$(\textbf{Seq})\ \frac{\langle s_0, \mathcal{S}\rangle \to_s \mathcal{S}' \qquad \langle s_1, \mathcal{S}'\rangle \to_s \mathcal{S}''}{\langle s_0\texttt{;}\ \ s_1, \mathcal{S}\rangle \to_s \mathcal{S}''} \qquad (\textbf{Ret})\ \frac{\langle s, \mathcal{S}\rangle \to_s \sum_j \pi_j \cdot \mathcal{N}(\boldsymbol{\mu}_j, \Sigma_j)}{\langle s\texttt{; return}\ \boldsymbol{X}, \mathcal{S}\rangle \to_p \sum_j \pi_j \cdot \mathcal{N}((\boldsymbol{\mu}_j)_{\boldsymbol{X}}, (\Sigma_j)_{\boldsymbol{X}})}$$

$$(\textbf{For-B})\ \frac{v_r \le 0}{\langle \texttt{for}\, x\, \texttt{in range}\, v_r \ \ s, \mathcal{S}\rangle \to_s \mathcal{S}} \qquad (\textbf{For-I})\ \frac{v_r > 0 \quad \langle s', \mathcal{S}\rangle \to_s \mathcal{S}' \qquad s' = s; x\texttt{=}x\texttt{+1; for}\, x\, \texttt{in range}\, v_r\texttt{-1}\ s}{\langle \texttt{for}\, x\, \texttt{in range}\, v_r \ \ s, \mathcal{S}\rangle \to_s \mathcal{S}'}$$

$$(\textbf{D-Asg})\ \frac{\langle e, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_i \quad \boldsymbol{\mu}_i' = \boldsymbol{\mu}_i[x = c_i] \quad \Sigma_i' = \Sigma_i[(x,x) = 0, (x,.) = \mathbf{0}^{\mathrm{T}}, (.,x) = \mathbf{0}]}{\langle x = e, \mathcal{S}\rangle \to_s \sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$(\textbf{P-Asg-Ind})\ \frac{\begin{array}{c}\langle e_j, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_{ij} \text{ for } j = 1,2 \qquad \boldsymbol{\mu}_i' = \boldsymbol{\mu}_i[X = c_{i1}] \\ \Sigma_i' = \Sigma_i[(X,X) = c_{i2}, (X,.) = \mathbf{0}^{\mathrm{T}}, (.,X) = \mathbf{0}] \qquad c_{i2} > 0\end{array}}{\langle X = \texttt{Normal}(e_1, e_2), \mathcal{S}\rangle \to_s \sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$(\textbf{P-Asg-Dep})$$
$$\frac{\begin{array}{c}\langle e_j, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_{ij} \text{ for } j = 1..3 \quad c_{i3} > 0 \quad \boldsymbol{\mu}_i' = \boldsymbol{\mu}_i[X = c_{i1}(\boldsymbol{\mu}_i)_Y + c_{i2}] \\ \Sigma_i' = \Sigma_i[(X,X) = c_{i1}^2(\Sigma_i)_{(Y,Y)} + c_{i3}, (X,.) = c_{i1}(\Sigma_i)_{(Y,.)}, (.,X) = c_{i1}(\Sigma_i)_{(.,Y)}]\end{array}}{\langle X = \texttt{Normal}(e_1 * Y + e_2, e_3), \mathcal{S}\rangle \to_s \sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$(\textbf{P-Cus-Asg})\ \frac{\begin{array}{c}\langle e_h, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_{ih} \text{ for } h = 1..n \\ \texttt{match}_2(\texttt{D}(c_{i1}, \ldots, c_{in})) = \sum_k \pi_{ik}^D \mathcal{N}(\boldsymbol{\mu}_{ik}^D, \Sigma_{ik}^D) \quad \pi_j' = \pi_i \pi_{ik}' \\ \boldsymbol{\mu}_j' = \boldsymbol{\mu}_i[X = \boldsymbol{\mu}_{ik}^D] \qquad \Sigma_j' = \Sigma_i[(X,X) = \Sigma_{ik}^D, (X,.) = \mathbf{0}^{\mathrm{T}}, (.,X) = \mathbf{0}]\end{array}}{\langle X = \texttt{D}(e_1, \ldots, e_n), \mathcal{S}\rangle \to_s \sum_j \pi_j' \cdot \mathcal{N}(\boldsymbol{\mu}_j', \Sigma_j')}$$

$$(\textbf{P-Op-PM})\ \frac{\begin{array}{c}\oplus \in \{+,-\} \quad \langle e, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_i \quad \boldsymbol{\mu}_i' = \boldsymbol{\mu}_i[X = (\boldsymbol{\mu}_i)_Y \oplus c_i] \\ \Sigma_i' = \Sigma_i[(X,X) = (\Sigma_i)_{(Y,Y)}, (X,.) = (\Sigma_i)_{(Y,.)}, (.,X) = (\Sigma_i)_{(.,Y)}]\end{array}}{\langle X = Y \oplus e, \mathcal{S}\rangle \to_s \sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$(\textbf{P-Op-MD})\ \frac{\begin{array}{c}\oplus \in \{*,/\} \quad \langle e, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_i \quad \boldsymbol{\mu}_i' = \boldsymbol{\mu}_i[X = (\boldsymbol{\mu}_i)_Y \oplus c_i] \\ \Sigma_i' = \Sigma_i[(X,X) = c_i^2 \oplus (\Sigma_i)_{(Y,Y)}, (X,.) = c_i \oplus (\Sigma_i)_{(Y,.)}, (.,X) = c_i \oplus (\Sigma_i)_{(.,Y)}]\end{array}}{\langle X = Y \oplus e, \mathcal{S}\rangle \to_s \sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$(\textbf{P-Sum})\ \frac{\begin{array}{c}\boldsymbol{\mu}_i' = \boldsymbol{\mu}_i[X = (\boldsymbol{\mu}_i)_Y + (\boldsymbol{\mu}_i)_Z] \\ \Sigma_i' = \Sigma_i[(X,X) = (\Sigma_i)_{(Y,Y)} + (\Sigma_i)_{(Z,Z)} + (\Sigma_i)_{(Y,Z)} + (\Sigma_i)_{(Z,Y)}] \\ \Sigma_i' = \Sigma_i[(X,.) = (\Sigma_i)_{(Y,.)} + (\Sigma_i)_{(Z,.)}, (.,X) = (\Sigma_i)_{(.,Y)} + (\Sigma_i)_{(.,Z)}]\end{array}}{\langle X = Y + Z, \mathcal{S}\rangle \to_s \sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$\boldsymbol{\mu}' = \begin{bmatrix} 15 \\ 20 \\ 0 \end{bmatrix} \quad \Sigma' = \begin{bmatrix} 2\ 0\ 0 \\ 0\ 0\ 0 \\ 0\ 0\ 0 \end{bmatrix}$$

The gray zeros correspond to the entries of the uninitialized Z variable. The zeros in the covariance coefficients for X and x mean that these variables are independent. Finally, the last probabilistic statement , according to P- Asg- Dep, results in

$$\boldsymbol{\mu}'' = \begin{bmatrix} 15 \\ 20 \\ 2 \cdot 15 \end{bmatrix} = \begin{bmatrix} 15 \\ 20 \\ 30 \end{bmatrix}$$
$$\Sigma'' = \begin{bmatrix} 2 & 0 & 2 \cdot 2 \\ 0 & 0 & 2 \cdot 0 \\ 2 \cdot 2 & 2 \cdot 0 & 2^2 \cdot 2 + 1 \end{bmatrix} = \begin{bmatrix} 2\ 0\ 4 \\ 0\ 0\ 0 \\ 4\ 0\ 9 \end{bmatrix}$$

Here we observe that the covariance between Z and X is updated with a non-zero value due to the dependency between variables, but the coefficients of x, Z are 0 as these variable remain independent; in fact, Dirac delta distributions always remain independent due to their 0 variance. Now suppose that the program is extended with $x =$ `Bernoulli`(0.5). In this case we apply P- Cus- Asg. First, we match the Bernoulli distribution with a GM having same mean and variance. Since a Bernoulli distribution is a discrete distribution, we set $\texttt{match}_2(D) = p \cdot \mathcal{N}(1, 0) + (1 - p) \cdot \mathcal{N}(0, 0)$—this ensures for any $D \sim$ `Bernoulli`(p) that i) and ii) are trivially satisfied. Then, we create two new mean vectors $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, setting, respectively, the mean of x equal to 1 and 0. Analogously we create two new covariance matrices Σ_1 and Σ_2 by setting to zero the row and column corresponding to

Fig. 4 Operational Semantics rules for conditional program statements; $\mathcal{S}$ stands for $\sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$

$$(\textbf{P-}\delta\textbf{-Cond})\ \frac{\pi_i' = \Pr_i(\varphi)\pi_i / \sum_i p_i(\varphi)\pi_i \qquad \varphi = (x \oplus e) \qquad \oplus \in \{<, >, \le, \ge, =, \neq\}}{\langle \texttt{condition}(\varphi), \mathcal{S}\rangle \to_s \sum_i \pi_i' \cdot \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)}$$

$$(\textbf{P-Eq-Cond})\ \frac{\begin{array}{c}\langle e, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_i \quad \boldsymbol{\mu}_i = \begin{bmatrix}(\boldsymbol{\mu}_i)_{\boldsymbol{A}} \\ (\boldsymbol{\mu}_i)_X\end{bmatrix} \ \Sigma_i = \begin{bmatrix}(\Sigma_i)_{\boldsymbol{A}} & (\Sigma_i)_{(.,X)} \\ (\Sigma_i)_{(X,.)} & (\Sigma_i)_{(X,X)}\end{bmatrix} \\ \boldsymbol{\mu}_i' = \boldsymbol{\mu}_i[\boldsymbol{A} = (\boldsymbol{\mu}_i)_{\boldsymbol{A}} + (c_i - (\boldsymbol{\mu}_i)_X)/(\Sigma_i)_{(X,X)}(\Sigma_i)_{(.,X)}, X = c_i] \\ \Sigma_i' = \Sigma_i[\boldsymbol{A} = (\Sigma_i)_{\boldsymbol{A}} - 1/(\Sigma_i)_{(X,X)}(\Sigma_i)_{(.,X)}(\Sigma_i)_{(X,.)}] \\ \Sigma_i' = \Sigma_i[(X,X) = 0, (X,.) = \mathbf{0}^{\mathrm{T}}, (.,X) = \mathbf{0}] \qquad \varphi = (X = e)\end{array}}{\langle \texttt{condition}(\varphi), \mathcal{S}\rangle \to_s \sum_i \pi_i \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$(\textbf{P-Tru-Cond})$$

$$\frac{\begin{array}{c}\varphi = (X \odot e) \quad \odot \in \{\le, \ge\} \quad \langle e, \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)\rangle \to_e c_i \\ \boldsymbol{\mu}_i = \begin{bmatrix}(\boldsymbol{\mu}_i)_{\boldsymbol{A}} \\ (\boldsymbol{\mu}_i)_X\end{bmatrix} \quad \Sigma_i = \begin{bmatrix}(\Sigma_i)_{\boldsymbol{A}} & (\Sigma_i)_{(.,X)} \\ (\Sigma_i)_{(X,.)} & (\Sigma_i)_{(X,X)}\end{bmatrix} \\ P_i = \begin{cases}\Phi(c_i \mid (\boldsymbol{\mu}_i)_X, (\Sigma_i)_{(X,X)}) & \text{if } \le \\ 1 - \Phi(c_i \mid (\boldsymbol{\mu}_i)_X, (\Sigma_i)_{(X,X)}) & \text{if } \ge\end{cases} \quad p_i = \begin{cases}-\phi(c_i \mid (\boldsymbol{\mu}_i)_X, (\Sigma_i)_{(X,X)}) & \text{if } \le \\ \phi(c_i \mid (\boldsymbol{\mu}_i)_X, (\Sigma_i)_{(X,X)}) & \text{if } \ge\end{cases} \\ \tilde{\boldsymbol{\mu}}_i = (\boldsymbol{\mu}_i)_{\boldsymbol{A}} + \frac{c_i - (\boldsymbol{\mu}_i)_X}{(\Sigma_i)_{(X,X)}}(\Sigma_i)_{(.,X)} \quad C_i = (P_i \cdot I)\left[(\cdot, X) = P_i I_{(\cdot,X)} + p_i \begin{bmatrix}\tilde{\boldsymbol{\mu}}_i \\ c_i\end{bmatrix}\right] \\ \pi_i' = P_i \pi_i \quad \boldsymbol{\mu}_i' = \boldsymbol{\mu}_i + \frac{p_i}{P_i}\Sigma_i \boldsymbol{e}_X \quad \Sigma_i' = \boldsymbol{\mu}_i(\boldsymbol{\mu}_i')^T + \frac{\Sigma_i C_i^T}{P_i} - \boldsymbol{\mu}_i'(\boldsymbol{\mu}_i')^T\end{array}}{\langle \texttt{condition}(\varphi), \mathcal{S}\rangle \to_s \sum_i (\pi_i' / \sum_j \pi_j') \cdot \mathcal{N}(\boldsymbol{\mu}_i', \Sigma_i')}$$

$$(\textbf{P-If})\ \frac{\langle \texttt{condition}(\varphi);\ s_0, \mathcal{S}\rangle \to_s \mathcal{S}' \qquad \langle \texttt{condition}(\neg\varphi);\ s_1, \mathcal{S}\rangle \to_s \mathcal{S}''}{\langle \texttt{if}\ \varphi : s_0\ \texttt{else}: s_1, \mathcal{S}\rangle \to_s \Pr(\varphi)\cdot\mathcal{S}' + (1 - \Pr(\varphi))\cdot\mathcal{S}''}$$

x. This yields:

$$\boldsymbol{\mu}_1 = \begin{bmatrix}15\\1\\30\end{bmatrix}, \quad \boldsymbol{\mu}_2 = \begin{bmatrix}15\\0\\30\end{bmatrix}, \quad \Sigma_1 = \begin{bmatrix}2&0&4\\0&0&0\\4&0&9\end{bmatrix},$$
$$\Sigma_2 = \begin{bmatrix}2&0&4\\0&0&0\\4&0&9\end{bmatrix}.$$

Finally, we derive the joint as $0.5 \cdot \mathcal{N}(\boldsymbol{\mu}_1, \Sigma_1) + 0.5 \cdot \mathcal{N}(\boldsymbol{\mu}_2, \Sigma_2)$. □

Two rules (P- Op- PM) and (P- Op- MD) define binary operations between random variables and values. When a value is added/subtracted to a random variable (P- Op- PM), the mean of the resulting random variable X is updated accordingly. Also, variable X inherits the variance and covariances of Y. For multiplication and division (P- Op- MD), the mean is updated as before, but the (co)variance are updated as a linear combination of the (co)variance of Y.

Example 5 Consider the program $X = \texttt{Normal}(1,1);\ Y = X+2;\ Z = Y*2$. After the first statement we have the state $\boldsymbol{\mu}_X = 1$ and $\Sigma_{(X,X)} = 1$ (P- Asg- Ind). At this point, all covariances equal 0 as we executed an independent assignment. The second statement updates the state according to P- Op- PM such that,

$$\boldsymbol{\mu}' = \begin{bmatrix}1\\3\\0\end{bmatrix} \quad \Sigma' = \begin{bmatrix}1&1&0\\1&1&0\\0&0&0\end{bmatrix}$$

The last statements updates the state according to P- Op- MD into

$$\boldsymbol{\mu}'' = \begin{bmatrix}1\\3\\6\end{bmatrix} \quad \Sigma'' = \begin{bmatrix}1&1&2\cdot1\\1&1&2\cdot1\\2\cdot1&2\cdot1&2^2\cdot1\end{bmatrix} = \begin{bmatrix}1&1&2\\1&1&2\\2&2&4\end{bmatrix}$$

The sum of two Gaussian random variables (P- Sum) updates the mean of the resulting random variable as the sum of the means of the operands. The covariance of the resulting random variable is the sum of the covariances of the operands with other variables, i.e., the new variable depends on all the variables that the operands depend on. The variance is the sum of the variances of the operands, and the covariances of the operands.

Example 6 Consider the program $X = \texttt{Normal}(15,2)$ $;\ Y = \texttt{Normal}(2,1);\ Z = X + Y$.

After the second assignment we have (P- Asg- Ind)

$$\mu' = \begin{bmatrix} 15 \\ 2 \\ 0 \end{bmatrix} \quad \Sigma' = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Thus, the third assignment updates the state according to P- SUM as

$$\mu'' = \begin{bmatrix} 15 \\ 2 \\ 15+2 \end{bmatrix} = \begin{bmatrix} 15 \\ 2 \\ 17 \end{bmatrix}$$

$$\Sigma'' = \begin{bmatrix} 2 & 0 & 2+0 \\ 0 & 1 & 0+1 \\ 2+0 & 0+1 & 2+1+0+0 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 2 \\ 0 & 1 & 1 \\ 2 & 1 & 3 \end{bmatrix}$$

There are three rules for conditioning: one for discrete distributions (P- δ- COND), and two for continuous (Gaussian) distributions (P- EQ- COND and P- TRU- COND). Conditioning on discrete distributions simply selects the Gaussian components that satisfy the condition φ. To this end, it selects the mixtures weights π_i for the components that satisfy φ and normalizes them. The function $\Pr_i(\varphi)$ computes the (0-1) probability of φ in the multivariate Gaussian distribution of the ith component.

Example 7 Consider a state $\mathcal{S}$ consisting of more than one component, $\sum_{i=1}^{3} \pi_i \cdot \mathcal{N}(\mu_i, \Sigma_i)$, where $\pi_1 = 1/2$, $\pi_2 = \pi_3 = 1/4$ and $(\mu_1)_x = 1$, $(\mu_2)_x = 2$ and $(\mu_3)_x = 3$. Then, according to P- δ- COND $\langle \texttt{condition}(x < 3), \mathcal{S}\rangle$ results in a state

$$\frac{1 \cdot 1/2}{3/4} \cdot \mathcal{N}(\mu_1, \Sigma_1) + \frac{1 \cdot 1/4}{3/4} \cdot \mathcal{N}(\mu_2, \Sigma_2) + \frac{0 \cdot 1/4}{3/4} \cdot \mathcal{N}(\mu_3, \Sigma_3)$$

which equals $2/3 \cdot \mathcal{N}(\mu_1, \Sigma_1) + 1/3 \cdot \mathcal{N}(\mu_2, \Sigma_2)$. □

For conditioning an equality over a continuous (Gaussian) random variable (P- EQ- COND), we use Thm. 5 introduced in Sect. 2.2. As a result of conditioning, the observed variable becomes a Dirac delta distribution. Note that, despite P- EQ- COND applying to the last random variable in the Gaussian mixture components, we may perform an affine transformation using a permutation matrix that swaps the order of random variables. Thus, conditions may refer to any variable in the Gaussian mixture.

Example 8 Recall the final state of the program in Example 6. Suppose that we extend the program with the statement $\texttt{condition}(Z = 1)$. The vector A in P- EQ- COND is instantiated as $[X, Y]^{\mathrm{T}}$. Thus, the mean vector and covariance matrix for A are computed as

$$\begin{aligned} \mu_A &= \begin{bmatrix} 15 \\ 2 \end{bmatrix} + \frac{1-17}{3} \cdot \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 15 \\ 2 \end{bmatrix} \\ &\quad + \begin{bmatrix} -32/3 \\ -16/3 \end{bmatrix} = \begin{bmatrix} 13/3 \\ -10/3 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \Sigma_A &= \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} - \frac{1}{3} \cdot \begin{bmatrix} 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} - \frac{1}{3} \cdot \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 8/3 & -2/3 \\ -2/3 & 2/3 \end{bmatrix} \end{aligned}$$

as a consequence, the resulting state after conditioning is

$$\mu' = \begin{bmatrix} 13/3 \\ -10/3 \\ 1 \end{bmatrix} \quad \Sigma' = \begin{bmatrix} 8/3 & -2/3 & 0 \\ -2/3 & 2/3 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Recall that covariances may be negative as the covariance matrix is positive definite (cf. Sect. 2.2).

Finally, for conditioning inequalities over a continuous (Gaussian) random variable (P- TRU- COND), we truncate the corresponding marginal of the Gaussian mixture, and approximate it with a new Gaussian mixture. To perform the approximation, first we observe that a truncated Gaussian mixture can be written as a mixture of truncated Gaussian, by suitably rescaling the weights. Then, we compute the moments of each truncated Gaussian using the results of Kan and Robin [29], and substituting each truncated Gaussian with a Gaussian having same mean and covariance matrix. To write the formulas of Kan and Robin, we introduce the following notation: I denotes the identity matrix and e_X denotes the column vector having 0 in each coordinate and 1 in the coordinate indexed by X. Recall that $\phi(x|\mu, \Sigma)$, $\Phi(x|\mu, \Sigma)$ are the probability density function and the cumulative density function of a Gaussian with mean μ and covariance matrix Σ, respectively (Sect. 2.2). Example 9 shows an example of applying this rule; in combination with an if-statement.

If-statements split each component of the state's Gaussian mixture into two new components (P- IF). Each new component is weighted by the probability of the condition ($\Pr(\varphi)$) and its negation ($\Pr(\neg\varphi)$). The two new components are the Gaussian mixtures resulting from executing each branch of the if-statement; on a state where the condition has been applied (by executing $\texttt{condition}(\varphi)$). For P- TRU- COND the negation of $\geq$ is $\leq$ and vice versa; as otherwise the cumulative probability of this event is not well-defined. For discrete distributions, $\neg\varphi$ is defined as usual.

Example 9 Consider the program $X = \texttt{Normal}(0, 1)$; $\texttt{if}\ X \geq 0\texttt{:}\ y = 0\ \ \texttt{else:}\ y = 1$. After the first assignment we set $\mu'_X = 0$ and $\Sigma'_{(X,X)} = 1$

(P- ASG- IND) —the remaining elements are uninitialized (0). Applying P- IF we obtain two branches: `condition` $(X \geq 0)$; $y = 0$ and `condition`$(X \leq 0)$; $y = 1$. This yields a mixture $\pi_1 \cdot \mathcal{N}(\boldsymbol{\mu}''_1, \Sigma''_1) + \pi_2 \cdot \mathcal{N}(\boldsymbol{\mu}''_2, \Sigma''_2)$. Since $\Pr(X > 0) = 1/2$, we have that $\pi_1 = \pi_2 = 1/2$. To compute $\boldsymbol{\mu}''_i$ and Σ''_i, we execute both branches using in the state $1 \cdot \mathcal{N}(\boldsymbol{\mu}', \Sigma')$. Since the conditions are inequalities on Gaussian random variables, we apply the rule P- TRU- COND. The result of conditioning for this state is computed as:

$$\boldsymbol{\mu}''_i = \begin{bmatrix} 0 \\ p_i/P_i \end{bmatrix} \quad \Sigma''_i = \begin{bmatrix} 0 & 0 \\ 0 & 1 - (p_i^2/P_i^2) \end{bmatrix}$$

For the branch $X \geq 0$ we have $(\boldsymbol{\mu}'_1)_X = \sqrt{2/\pi}$ and for $X \leq 0$ we obtain $(\boldsymbol{\mu}'_2)_X = -\sqrt{2/\pi}$. The variance is the same for both branches: $(\Sigma'_1)_{(X,X)} = (\Sigma'_2)_{(X,X)} = 1 - (2/\pi)$. Note that we do not need to update weights because after normalization they remain as 1. Finally, the Dirac delta assignments update the mean of the mean vector as described above (D- ASG). The resulting mixture is:

$$1/2 \cdot \mathcal{N}\left(\begin{bmatrix} 0 \\ \sqrt{2/\pi} \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 - (2/\pi) \end{bmatrix}\right)$$
$$+1/2 \cdot \mathcal{N}\left(\begin{bmatrix} 1 \\ -\sqrt{2/\pi} \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 - (2/\pi) \end{bmatrix}\right)$$

3.2 Properties of Gaussian Mixture Semantics

In what follows, we show that programs invoking a subset of the operational semantics rules produce an exact posterior distribution, the complete set of rules produces an approximate posterior distribution, and also that the inference engine always terminates for well-formed programs. In a nutshell, programs that do not make use of the rules (P- CUS- ASG) on continuous distributions or (P- TRU- COND), produce an exact posterior. If a program uses the previous rules, the result is an approximate posterior distribution.

Given a state $\mathcal{S}$, we say that the Gaussian mixture resulting from applying a rule is an *exact posterior* iff it is a transformation obtained by applying the theorems for multivariate Gaussian distributions lifted to Gaussian mixtures (Sect. 2) or it respects the standard properties of probability measures.

Proposition 1 *For any state $\mathcal{S}$, applying any of the rules* (P- IF), (P- δ- COND) *or* (P- CUS- ASG) *in the case of discrete distributions, results in an exact posterior $\mathcal{S}'$.*

Proof sketch The proposition follows by lifting the lemmas we proved in our previous work [13] to Gaussian mixtures (Sect. 2.3). The rule P- IF is an instance of the additivity property of probability measures. Given two disjoint events, in our semantics φ and $\neg\varphi$, the probability mass for the union of these events is the sum of the probability mass of each event. In our semantics, P- IF represents the union of the events φ and $\neg\varphi$. The same reasoning holds for P- CUS- ASG applied to assigning a discrete distribution, as discrete distributions are encoded as a weighted sum over their set of (disjoint) outcomes. The rule P- δ- COND is an instance of redistribution of mass by applying Bayes rule using a 0-1 likelihood function $\Pr_i(\varphi)$.

Similar to above, given a state $\mathcal{S}$, we say that the Gaussian mixture resulting from applying a rule is an *approximate posterior* iff it is a transformation that approximates the exact posterior with a Gaussian mixture.

Proposition 2 *For any state $\mathcal{S}$, applying rule* (P- CUS- ASG) *or rule* (P- TRU- COND)*, results in an approximate posterior $\mathcal{S}'$ matching up to second order moments.*

Proof sketch For (P- CUS- ASG) the proposition follows by definition: the $\texttt{match}_2$ function is required to set the mean and variance (1st and 2nd moments) of X in $\mathcal{S}'$ to the mean variance of the distribution D. Rule (P- TRU- COND) is an instance of the results by Kan and Robotti [29]. Specifically, it is the case for finding the mean and variance of a truncated Gaussian for a marginal of a multivariate Gaussian distribution lifted to Gaussian mixtures. Thus, the proposition holds for P- CUS- ASG as well.

Propositions 1 and 2 categorize the class of programs for which we can compute exact or approximate posterior distributions. In Sect. 4, we show how to analyze different types of privacy protection mechanisms using both exact and approximate posterior distributions.

Proposition 3 *(Termination) Given a well-formed program, the process of computing the resulting Gaussian mixture always terminates.*

Proof sketch Well-formed programs are unbounded but finite sequences of program statements. Thus, to prove termination, it suffices to prove that each program statement is evaluated in finite time. Termination for all the rules except (P- IF), (P- δ- COND), (P- TRU- COND) and (P- CUS- ASG) follows from the results in our previous work [13] and Gaussian mixtures being a finite sum over mixture components. In our previous work [13], we proved that applying these transformations (excluding the ones previously mentioned) on a multivariate Gaussian distribution terminates for all rules. The version of the rules introduced in this paper apply the same transformations to each component of the Gaussian mixture. Since there is a finite number of components, it holds that this process terminates for these rules. If-statements (P- IF) are resolved in constant time. The rule P- δ- COND normalizes the weight for each Gaussian component π. The normalization process requires making a probability query on the multivariate Gaussian and compute the total probability mass of the components that satisfy the condition, both operations are computed in finite time. The rule P- TRU- COND executes a series of matrix operations and queries to probability/cumulative density functions. These operations are completed in finite time. Finally, custom assignments P- CUS- ASG require computing the first two moments of the custom distribution D. The function $\texttt{match}_2$ described in Sect. 3.1 is required to finish in finite time, and setting the moments in the resulting Gaussian mixture is performed in constant time.

The implementation of the inference engine presented in this paper not only terminates for all well-formed programs, but it can efficiently analyze systems with thousands of random variables (see Sect. 4.4).

4 Experimental Evaluation

We evaluate the presented method by implementing it in a prototype tool. We use the tool to analyze four different privacy protection mechanisms: randomized response, differential privacy with the Gaussian mechanism, generalization (as used with k-anonymity), and differential privacy with the Laplacian mechanism. The first two cases demonstrate that non-trivial programs can exhibit information flow semantics that is captured exactly by Gaussian semantics. This allows our method to symbolically compute the representation of attacker beliefs precisely. The latter two examples cannot be represented faithfully with Gaussian mixtures up-to two moments precision. We exploit them to investigate the strength of the approximation with Gaussian mixtures. Finally, we briefly report on the scalability of our tool.

4.1 Implementation

In the following sections we refer to the implementation of the Privug method for a subset of Python as Privugger. We implement a Gaussian mixture version of Privugger on top of a a slightly modified version of the inference engine SOGA [17]. To distinguish the version of Privugger using SOGA from the previous version based on multivariate Gaussian distributions, we refer to the previous as Privugger-exact. SOGA implements the Gaussian mixture semantics presented in Sect. 3.1. The core of the implementation is quite simple: it mostly translates the input Python programs to the language accepted by SOGA.@ Then we use SOGA for privacy analysis, following the Privug method. The small modifications to SOGA were required in the parser front-end to capture the subset of Python programs which can be generated from the syntax of Sect. 3. The transformation is implemented using the language parser generator infrastructure provided by ANTLR [30].@ The modified tool is an open source project and can be found at https://github.com/RasmusCarl/Privug-SOGA, together with the source code for the experiments below.

```
1 value = bern(0.5)    # Prior attacker
2     knowledge
3 coin1 = bern(0.5)
4 coin2 = bern(0.5)
5 if coin1 == 1:
6     output = value
7 else:
8     output = coin2
9 observe(output == 1)   # Attacker observation
```

Listing 3 The randomized response case program.

4.2 Case Studies with Exact Gaussian Mixture Semantics

All cases analyzed in this section, can be analyzed with our symbolic tool precisely, as they only involve manipulating Gaussian and discrete distributions and do not condition on continuous random variables.

Randomized Response

Randomized response is a method used to disclose results of surveys concerning sensitive issues without disclosing the truthfulness of the participants [18]. This mechanism has a myriad of applications where the answer to the question may involve sensitive information about the participant—e.g, in healthcare surveys regarding medical conditions or unhealthy or culturally controversial lifestyle habits. The core idea is to return a random answer for a participant. We consider a version of the method implemented using a fair coin, shown in Lst. 3. For the illustrative purposes, suppose that `value` contains the answer to the question *"Do you have medical condition X?"*. Line 1 models the attacker prior knowledge, i.e., the respondent's reply may be yes (1) or no (0) with probability 0.5. This models an attacker with no prior knowledge about the respondent's medical condition. Before giving her response, the respondent flips a fair coin: if the coin gives head, she answers truthfully (lines 5–6), otherwise her answer is random, using a second coin toss (lines 7–8). Knowing that the released answer is 1, an attacker can infer the exact probability that the respondent has answered yes, by deriving the posterior distribution over `value`.

The randomized response program uses Bernoulli distributions, which are discrete distributions with support $\{0, 1\}$ and therefore can be encoded as degenerate Gaussian mixtures. In particular, a Bernoulli distribution with parameter p can be encoded as the Gaussian mixture $p \cdot \mathcal{N}(1, 0) + (1 - p) \cdot \mathcal{N}(0, 0)$. Our engine performs exact inference for this program (see Prop. 1), yielding a posterior distribution for `value`, a Bernoulli distribution with $p = 0.75$.

```
1  array[10] male_2130
2  array[10] male_3140
3  array[10] male_4150
4  array[10] male_5160
5  array[10] female_2130
6  for i in range(10):
7  # declare data  arrays (secrets)
8  # Prior attacker knowledge
9      male_2130[i] = gauss(465000, 100000)
10     male_3140[i] = gauss(465000, 100000)
11     male_4150[i] = gauss(465000, 100000)
12     male_5160[i] = gauss(465000, 100000)
13     female_2130[i] = gauss(465000, 100000)
14 for i in range(10):
15    total_all = female_2130[i]+male_2130[i]
16         +male_3140[i]+male_4150[i]
17         +male_5160[i]
18 total_all = 0.02 * total_all
19 # Attacker observation
20 observe(total_all == 508389.1)
```

Listing 4 Computing average income across age groups and genders

Differential Privacy with the Gaussian Mechanism
We analyze a program computing statistics on a database containing incomes for different genders and age groups. Average income data for broad population are available to attackers through public national statistics banks [31–33]. Leakage of private information and database reconstruction attacks are known issues (e.g., in US census data [34]). We use our inference engine to quantify the increase of attacker knowledge, as she gradually obtains statistics from a database. The case study uses a small database, but in Sect. 4.4 we show that our inference engine scales to databases with thousands of individuals.

Consider a data analyst that releases average statistics on population income for different age groups and genders. An attacker with access to the statistics attempts to learn the income of an individual in the database. We consider the synthetic dataset shown in Tbl. 1. The table shows the income for individuals in different age groups and genders. The data is constructed by sampling different distributions for each age group. We consider three different cases.

Case 1: The attacker obtains the average income for the entire dataset. *Case 2:* The attacker also obtains the average income for all males. *Case 3:* The attacker also obtains the average income for all males in the age group 21–30.

In all cases the attacker attempts to learn the income of the first male in database in the age group 21–30. For all incomes the prior is set to $\mathcal{N}(465000, 100000)$. This choice is meant to mimic an attacker that has access to the average income of an entire population, but not for the age groups.

To understand the privacy risks for these three cases we use the Privug method. In the following code snippet we show how to model the steps for Case 1 using our inference engine. The code for the other cases follows a similar structure. We first model the program (Step 2, see Lst. 4). In lines 1–5 the arrays containing the priors of the income for the individuals in the database Tbl. 1 are declared (they contain secret values). In lines 7–12, the entries in the array are set to the prior distribution which represent the possible incomes that the attacker considers possible before making any observations (step 1). The victim is the first male in the 21–30 age group. We denote his prior $\Pr(I_1)$ in the formalization below. In lines 14–16, we compute the average income of each group, which determines a probability of outcome $\Pr(O \mid I_i)$. In Line 18, we add the attacker observation in the observe statement (step 3). The observed value is the actual mean from the secret data set shown in Tbl. 1. For cases 2-3 we extend the program to calculate and observe the mean also for the gender group (Case 2) and for the gender and age group (Case 3). In both cases, we calculate the observed values from the secret non-disclosable data of Tbl. 1.

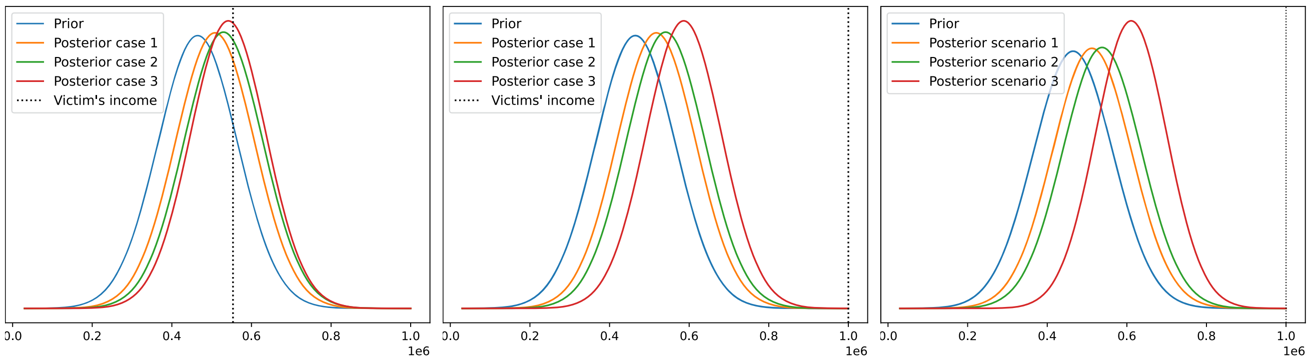
Figure 5a shows how attacker knowledge is updated in the three cases above, and how close it is to real victim data (vertical line). We plot the prior attacker knowledge $\Pr(I_1)$, and for each case we plot the posterior distribution after conditioning on the output $\Pr(I_1 \mid O)$. As the plot shows, the attacker knowledge gets closer to the actual income when obtaining more information. The most accurate attacker knowledge is case 3 where the attacker obtains several average statistics, and the mean of the posterior gets rather close to the secret income.

The data analyst is also interested in understanding the privacy risk when the dataset contains an outlier. To analyze the privacy risk for that case we manually change the income of the victim to be 999999, which is a higher income than his peers in same age group. This change will lead to the attacker observing a higher value for the average incomes. Figure 5b shows that the posterior does not change much for cases 1–2, and only in Case 3 the posterior switches towards the outlier. Finally, the data analyst wants to know if making the outlier victim's group smaller compared to the other groups has an impact. For this we manually remove the five last entries in the age group 21–30 for males. The result is shown in Fig. 5c where it can be seen that the posterior of the victim only changes even more dramatically in the third case, as the mean for the group provides more information in a smaller group.

Given the above results, the data analyst decides to use a differentially private mechanism [3] to protect the individuals' privacy. For this case study we apply differential privacy to the case where the dataset contains an outlier and the age group 21–30 of males has been reduced to five people. Differential Privacy is used in realistic settings for the release of public statistics. Notably, it was used in the 2020 US Census as a result of privacy issues in previous US Census editions [34]. Intuitively, if the privacy protection mechanism satisfies differential privacy, then the impact of an individual on the output of the program is negligible. More precisely, differen-

Table 1 Income per year per age groups and gender, in DKK (the secret dataset)

Age Group	Income of Males									
21–30	553663	433856	713989	471166	435411	745048	577540	540576	442278	504165
31–40	429254	565260	490365	568815	446628	417811	403353	618130	454271	444181
41–50	612930	486621	535591	584664	467942	593904	577935	366869	614758	676634
51–60	421815	619666	723055	498242	540499	541263	494399	596725	474274	504156
Age Group	Income of Females									
21–30	570584	486264	433134	418759	222708	455372	457377	383811	394170	409574



(a) Basic case (b) Observed when the victim is an outlier (c) An outlier victim in a group reduced to 5 individuals

Fig. 5 The prior vs the attacker's belief after an observation (no differential privacy). The vertical line represents the secret income of the victim, the first individual in `male_2130` that the attacker is attempting to guess

tial privacy states that: a randomized mechanism $\mathcal{M} : \mathbb{I} \to \mathbb{O}$ is (ε, δ)-differentially private if for all outputs $\mathcal{O} \subseteq \mathbb{O}$, and any two neighboring inputs $i_1, i_2 \in \mathbb{I}$ (inputs differing by including or excluding one individual), the following holds:

$$\Pr(\mathcal{M}(i_1) \in \mathcal{O}) \quad \leq \quad \exp(\varepsilon) \Pr(\mathcal{M}(i_2) \in \mathcal{O}) + \delta \ . \qquad (3)$$

The neighboring relation between inputs depends on the input domain ($\mathbb{I}$). For instance, when it applies to datasets of n natural numbers, $\mathbb{N}^n$, it is usually defined as the first norm $||i_x - i_y||_1$. The parameter ε is often referred to as the *privacy parameter*, and it is used to specify the required level of privacy. Formally, ε captures the log-distance between the probability distributions in definition of differential privacy, see eq. 3.

Intuitively, the lower the value of ε the more similar the distributions are, and the harder it is for the attacker to distinguish what dataset was used to produce the output—thus, resulting in increased privacy for the individuals in the dataset. The parameter δ is the probability of failure. This parameter relaxes the definition of differential privacy. It is used to specify the probability that pure differential privacy (i.e., with $\delta = 0$) does not hold. This parameter may be used to, e.g., enable high utility gains while keeping a good level of privacy. Both ε and δ are often determined empirically [35].

We analyze a differentially private mechanism for the three cases presented above. To this end, we apply the Gaussian mechanism [3], which adds Gaussian noise to the observable output (o) as $o + \mathcal{N}(0, \sigma^2)$. The parameter σ^2 is calculated as follows:

$$\sigma^2 = 2\Delta^2 \log(1.25/\delta)/\varepsilon^2. \qquad (4)$$

Adding Gaussian noise is proven to satisfy (ε, δ)-differential privacy [3]. The sensitivity $\Delta \in \mathbb{R}$ denotes how much o changes if computed in two datasets differing in at most 1 entry. In our setting, it is $\Delta = (\max_{income} - \min_{income})/\text{size}_{DB}$. We set $\delta = 1/\text{size}_{DB}^2$—as usual for this query [3]. We try different values for ε, to model different levels of privacy protection. We remark that our method can be used to determine the values of ε and δ that satisfy high level privacy requirements. For instance, privacy requirements specified as probability queries for a given individual or using quantitative information flow metrics [12]. The program implementing the Gaussian mechanism is shown in the following listing

```
noise = gauss(0,4334263030.198959)
```

```
2 total_all = total_all + noise
3 # Attacker observation
4 observe(total_all==518697.27659574465)
```

We only show lines that change namely: line 1 where the noise distribution is defined, and line 2 where we add the noise to the output. The variance σ^2 of the noise distribution is calculated using eq. 4. In the example program above the variance of the noise distribution is calculated using $\varepsilon = 1$. The programs for the other cases are similar, but contain a separate noise distribution for each released average statistics.

The plot in Fig. 6 shows the updated attacker knowledge in the 3 cases. For each of the plots in the figure we used a different value for ε. We observe a decrease in privacy risks when using differential privacy; as, for some ε values, the change in attacker knowledge is insignificant for all cases.

This analysis illustrates the use of Privug to explore various values of ε.For instance, although values of ε lower than 1 are often used in practice [35], this analysis shows that (for this case study) larger values of ε result in a minor impact on attacker knowledge. Only for a very large value of ε we observed an update in attacker knowledge. Finally, as expected, the plot shows that the impact of the victim's data on the released statistics is minuscule compared to the non-differentially private version of the output shown in Fig. 5.

4.3 Approximate Gaussian Mixture Semantics

The case studies presented so far could be analyzed exactly using Gaussian Mixtures, due to the fact that only discrete and multivariate Gaussian distributions were involved. We now present two new cases in which multivariate Gaussian Mixtures are used to approximate the distributions in question.

Generalization

A common technique to anonymize sensitive data in a database is to substitute the exact values of some attributes with broader categories, for example ranges of values [19]. An example is shown in Fig. 7a, where the information about the census of an individual is only represented by a Boolean value; one if the census exceeds a given threshold and zero otherwise. We assume that the attacker models her prior knowledge as a Gaussian distribution (albeit the tool supports any priors approximated by a Gaussian mixture).

Our engine is not able to perform exact inference for the program of Fig. 7a. The reason is that when the if branch of the conditional statement is executed, the joint distribution is truncated to $\texttt{income} > 120000$. Since the prior distribution is a Gaussian, the resulting truncated distribution is a truncated Gaussian; not a Gaussian anymore. The same happens when the else branch is entered. The marginal distribution of `income` before the observe statement is shown in Fig. 8a. Notably, the exact distribution is given by a mixture of two truncated Gaussians, each corresponding to a different value of high. Our engine approximates each component with a Gaussian component having same mean and variance as the corresponding truncated Gaussian, yielding the marginal represented in blue. Adding the observation corresponds to selecting a single component. The corresponding posterior distributions are depicted in Fig. 8.

Interestingly, even though the posterior density is approximated, the moments of the posterior of `income` are computed exactly. This is due to the fact that the Gaussian components used in the approximation have the same moments as the truncated Gaussian components of the exact posterior. However, if we consider a more complex example involving nested if, such as the one shown in Fig. 7b, not only the posterior density, but also the posterior moments will be subject to an approximation. Figure 9 shows how the error in the posterior densities and in the posterior moments varies as the range of the categories varies, making the true posterior more difficult to be approximated by Gaussian components.

Differential Privacy with the Laplacian Mechanism

We consider again a differential privacy mechanism, this time adding a Laplacian noise to the observable output (o) as $o + \text{Laplace}(o, b)$. Here, b is calculated as $.{}^{\Delta}/_{\varepsilon}$ where $\Delta = \max_{\texttt{income}} / \text{size}_{DB}$ for $\varepsilon = 1$. Additive Laplacian noise is proven to satisfy (ε, δ)-differential privacy [35], but it represents a harder case for our tool, as it relies on a non-Gaussian noise.

The program is presented in Fig. 10a. We consider a database of 10 individuals, with uniform prior either between 200 and 400 (l. 6), or between 300 and 600 (l. 9). Again we compute the mean over all individuals, but this time we add a Laplacian noise with mean 0 and $b = 60$ before conditioning to possible observations. Computing the distribution up to this point is not possible using just multivariate Gaussians, because of the presence of non Gaussian distributions, i.e. the uniform priors and the Laplacian noise. However, for both we can define the operator $\texttt{match}_2$ as follows.

For uniform distributions, we the split the support of the uniform into two equal sub-interval, and for each sub-interval we take a Gaussian component with mixing coefficient 0.5 and mean and variance equal to those of a uniform distribution having as support the sub-interval. For Laplacian distributions the procedure is slightly more complex. Given a Laplacian distribution with parameters (μ, b) we look for a Gaussian mixture of two components in the form $p\mathcal{N}(\mu, \alpha b^2) + (1 - p)\mathcal{N}(\mu, k\alpha b^2)$ where $\alpha > 0, k > 1$. Equating the first four central moments of the Laplacian with those of the Gaussian Mixture we obtain the following parametric solution for the parameters of the GM: $p = \frac{8}{\alpha^2 - 4\alpha + 12}$, $k = \frac{2(\alpha-6)}{\alpha(\alpha-6)}$. It can be verified that with these definitions $\texttt{match}_2$ satisfies conditions i) and ii) of Section 3.1 (in the case of Laplacians it satisfies the conditions for any

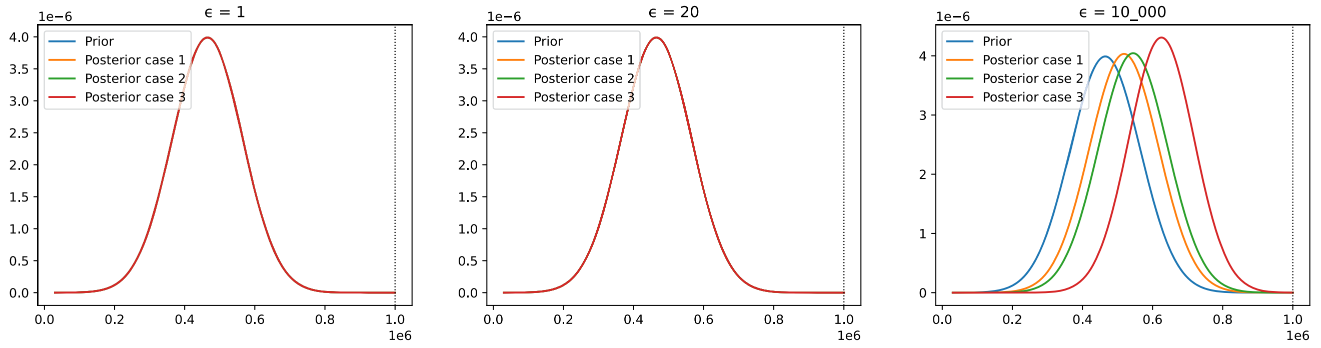


Fig. 6 Updated attacker knowledge after adding observations with differential privacy. Each plot has a different value for ε

Fig. 7 Example generalization programs

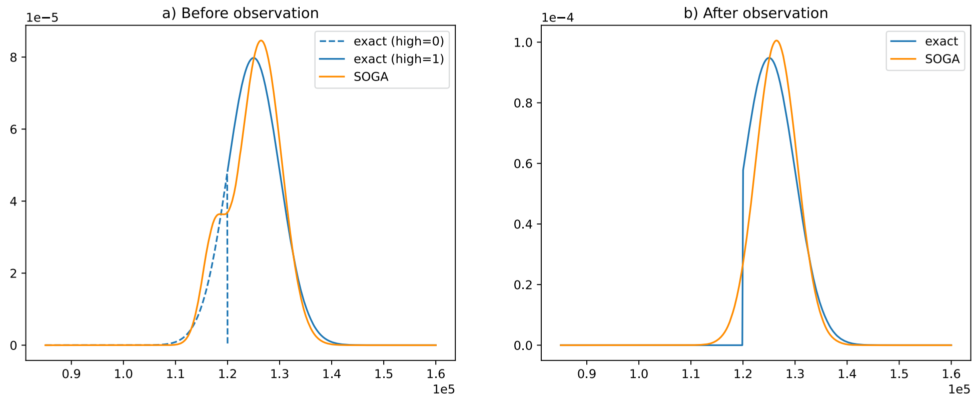
```
income = gauss(125000, 5000)
if income > 120000:
    high = 1
else:
    high = 0
observe(high == 1)
```

(a) Abstract to two classes

```
income = gauss(110000, 5000)
if income > u:
    high = 2
else:
    if income > l:
        high = 1
    else:
        high = 0
observe(high == 1)
```

(b) Abstract to three classes

Fig. 8 Marginal posterior over `income` for the program of Fig. 7a



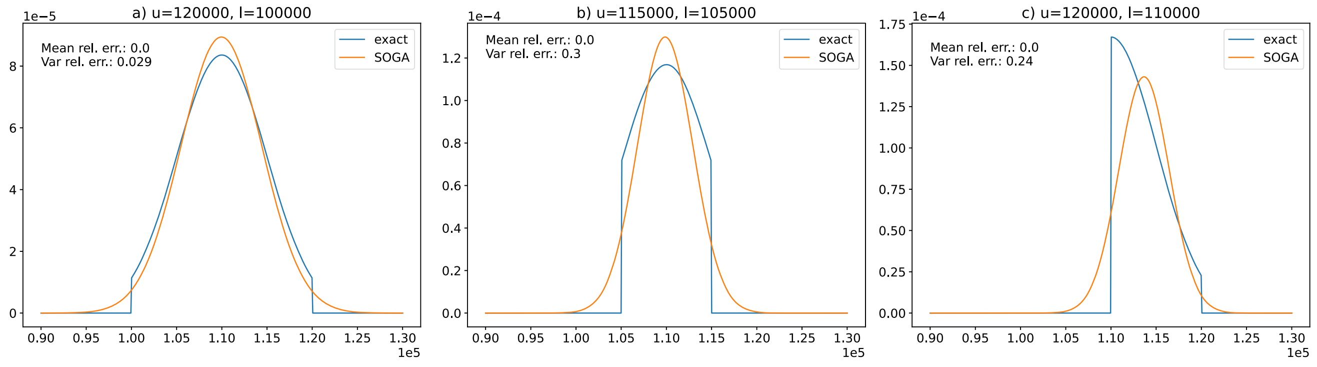


Fig. 9 Marginal posteriors over `income` and relative error of moments for the program of Fig. 7b as the bounds l, u vary

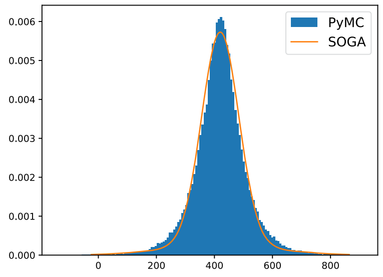
Fig. 10 Analysis of the Laplacian Mechanism for differential privacy

```
array[2] lower
array[8] higher
sum = 0

for i in range(2):
    lower[i] = uniform([200,400], 2)
    sum = sum + lower[i]
for i in range(8):
    higher[i] = uniform([300,600], 2)
    sum = sum + higher[i]

sum = 0.1*sum
noise = laplace(0, 60)
sum = sum + noise
```

(a) A program using the Laplacian mechanism



(b) Output distribution of program in Fig. 10a.

α). In the example, we used a numerical optimizer to find the value of α that minimizes the KL divergence between the Laplacian and the Gaussian Mixture. The optimization takes around 0.7 seconds.

Using the approximations, we compare the distribution carried by program in Fig. 10a, computed either by SOGA or by taking 80000 samples in PyMC [36]. Figure 10b shows that despite the approximations, the distribution is captured fairly well by the Gaussian mixtures. This is confirmed by the moments computation, where we computed a relative error of 0.00024 for the mean and of 0.00369 for the variance.

4.4 Scalability

We now evaluate the scalability of the new exact inference engine based on SOGA with semantics based on Gaussian mixtures. We evaluate two key aspects of our inference engine, focusing on scalability with respect to the number of variables and the number of components, as these are the primary factors impacting performance. In the first experiment, we assess scalability by increasing the number of variables in the program, both with and without conditioning on observed values. Increasing the number of variables raises the dimensionality of the Gaussian required to approximate the distribution. The second experiment evaluates scalability with respect to the number of components in the Gaussian mixture, by using a sequence of if statements, where each additional if statement doubles the number of components, resulting in exponential growth.

Scalability with Respect to the Number of Variables

We consider two synthetic benchmarks which were originally used to measure the scalability of Privug with two different inference engines backends: one based on the Monte-Carlo inference [12] and one based on the exact inference using multivariate Gaussian distributions [37], but not mixtures.

We compare the new inference engine with the one based on multivariate Gaussian distributions, as both are capable of producing exact results of the posterior distribution. The first benchmark computes the sum over an increasing number of variables: $O = \sum_{i=1}^{n} X_i$ with $X_i \sim \mathcal{N}(1, 1)$ for increasing values of n. The second program performs the same computation but adds a condition statement $\texttt{condition}(O = c)$. These benchmarks are chosen since the scalability of Bayesian inference engines mainly depends on the number of random variables. The purpose of evaluation scalability is to consider a more realistic setting for the case study in sections. 4.2 and 4.3, since real world datasets might contain many individuals. We evaluate up to $n = 6000$ variables; which is sufficient to observe the performance difference with respect to Privugger-exact. Figure 11 shows the measured times for the two benchmarks. In both benchmarks a similar pattern occurs where the SOGA-based engine scales worse as the number of random variables is increased. The plots shows that for an increasing number of variables the difference in the log-scale plot stays constant, which translates to an order of magnitude increase in the gap in running times. Up to around 2000 variables the running time for SOGA is negligible, but above that the running time is becoming an obstacle for practical application. In comparison the running time is still negligible for 6000 variables with Privugger-exact. SOGA demonstrates better scalability than the MCMC based Privugger [12] and the PSI engine [38]. In our previous work [13], we reported lower scalability of these engines compared to Privugger-exact with only 700 variables. The increase in running times for SOGA with respect to Privugger-exact is likely caused by the specific implementation details, and there is ongoing work to make the gap in performance smaller. However, using SOGA allows analyzing a larger and more expressive subset of Python programs and the scalability benchmarks shows that it is possible to compute an exact posterior in programs with thousands of random variables which makes it a practical inference engine for privacy risk quantification using Bayesian inference. This was not possible with the previous version of Privug, not using mixtures.

Scalability with Respect to the Number of Mixture Components

To evaluate the scalability with respect to the number of components needed to approximate the program distribution, we consider a variation of the program presented in Figure 7a, where we vary the number of incomes considered. The new program is reported in Figure 12a. For each iteration of the loop, the presence of a conditional statement doubles the number of components used to approximate the distribution. Therefore, the number of components grows exponentially in the number of if statements present in the program. Figure 12b shows the growth of the computational times with respect to the number of components (in log scale) used to represent the final distribution, for ten different values of N. Since if statements were not supported in Privug's syntax, this experiment does not include a comparison.

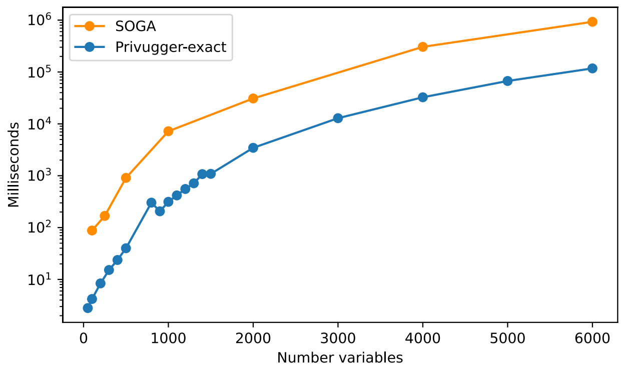
5 Related Work

This work constitutes an extension of our previous work [13]. In that work, we presented a method for privacy risk quantification using exact Bayesian inference. The method targeted a subset of Python, and used an exact inference engine based on multivariate Gaussian distributions for the semantics of the language. This choice of semantics limited the expressiveness of the language; notably if-statements and discrete distributions were not supported. In this work, we present an extension of the language and provide a semantics based on Gaussian mixtures. This extension makes it possible to perform exact reasoning on programs with discrete random variables, and approximate reasoning for non-Gaussian random variables. It also enables support for programs with conditional statements (both on discrete and continuous random variables), and, importantly, it supports if-statements. These extensions make it possible to analyze a much wider range of programs. Compared to [13], we provide new case studies that demonstrate a new set of programs for which we can analyze privacy risks. These new case studies include privacy preserving mechanisms such as randomized response, attribute generalization, and the Laplacian mechanisms to ensure differential privacy, which could not be analyzed in the previous work. Finally, we compare the scalability of the inference engine to that of the engine in the previous work. The evaluation shows that the scalability of our engine is worse than in our previous work; which is expected due to the wider range of programs supported. Yet the evaluation shows that our new engine can be used to analyze systems with thousands of random variables, making it a applicable in practical applications.

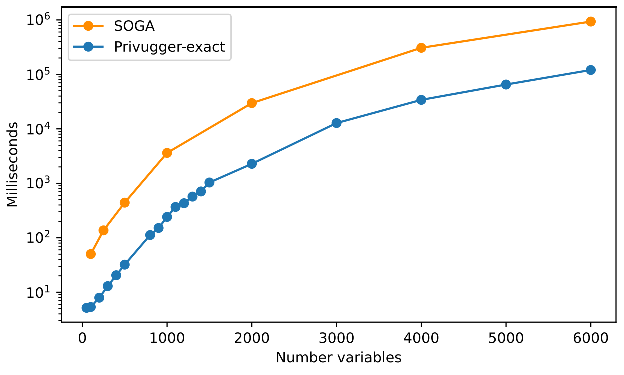
Table 2 shows an overview of related works as well as different Privug inference engines. The comparison focuses on the main features of the inference engines behind the methods; as these features determine the type of programs that can be analyzed. It compares whether the method supports discrete and/or continuous random variables. This feature determines the flexibility in modeling attacker knowledge. We state whether the method computes exact results, approximations or a combination of both. Table 2 shows the input for each method. This is important as it determines whether the method can be directly applied to source code or it needs to be converted to a custom language or the program must be executed repeatedly to obtain input/output samples from it. Finally, the table indicates the result of the analysis for each method. It can either be information leakage measures or the entire posterior attacker knowledge. Prior and posterior attacker knowledge can be used to compute most information leakage measures; and, in particular, it can be used to compute any of the metrics mentioned in the table. In what follows we discuss the methods in Table 2 and other related works in the domain of semantics of probabilistic programs.

The majority of existing methods to estimate privacy risks use sampling based techniques [5, 6, 10–12, 39]. In [12], Privug made use of MCMC algorithms to perform Bayesian inference, e.g., *Metropolis-Hastings* or *Hamiltonian Monte Carlo* [40–42]. Other sampling based methods target specific quantitative information flow metrics [4]—these metrics are supported by Privug as well [12]. LeakWatch/Leakiest [5, 6] use program samples to estimate mutual information between secret inputs and public outputs. Cherubin et al. and Romanelli et al. [10, 11], use machine learning to compute metrics from the g-leakage family [4]. These methods treat programs as black-boxes, so they can analyze any program, as opposed to our method that targets a subset of Python programs. However, their accuracy guarantees are proven in the limit, i.e., assuming an infinite size sample. In practice, samples are finite and it is often difficult to ensure that results are accurate; specially for programs with large number of variables (such as the ones in Sect. 4.4). In contrast, our inference engine produces exact results for an expressive subset of Python. This an important factor, as under-approximations could miss important privacy breaches. We demonstrate in Sect. 4.2 how to perform exact analyses for randomized response and the Gaussian mechanism to ensure differential privacy; two widely used privacy protection mechanisms. Our engine also computes approximate results (cf. Sect. 4.3), in those cases we showed that we obtain similar accuracy than the MCMC-based Privug.

There exist several works that use exact inference in the context of privacy risk analysis. SPIRE [9] uses the exact inference engine PSI [38, 43] to model attacker knowledge



(a) Without conditioning



(b) With conditioning

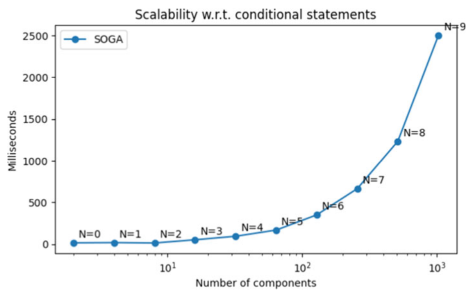
Fig. 11 Running times (milliseconds in log-scale) for privugger engines on increasingly large input models $\sum_i^n X_i$

Fig. 12 Running times for SOGA on increasingly larger number of conditional statements

```
array[N] income
array[N] high

for i in range(N):
    if income[i] > 120000:
        high[i] = 1
    else:
        high[i] = 0
```

(a) Generalized two-classes abstraction



(b) Computational time

and synthesize privacy enforcers. PSI computes a symbolic representation of the joint probability distribution of a given program. It can handle continuous and discrete random variables. It targets a more expressive programming language than the subset of Python that our engine supports. However, in our previous work we showed that PSI scales poorly compared to the multivariate Gaussian engine for Privug [13]. The scalability evaluation (cf. Sect. 4.4)—which uses the same benchmark programs as in [13]—shows that our new engine scales well for programs with thousands of random variables—this was not the case for PSI in [13]. Hakaru [44] and SPPL [45] are exact inference engines—not used for privacy risk analysis. These engines do not handle some features of our language. Hakaru cannot handle conditioning probability-zero events (as P- Eq- Cond in Fig. 4). SPPL does not support linear combination and sum of Gaussians (as P- Asg- Dep and P- Sum in 3). QUAIL [7] computes mutual information between input and output variables. It performs forward state exploration of a program to construct a Markov chain, which is then used to compute mutual information. HyLeak [8] is an extension of QUAIL that perform uses sampling to improve performance; which, consequently does not produce exact results. QUAIL and Hyleak do not support conditioning (neither P- Eq- Condnor P- δ- Condnor P- Tru- Cond). QUAIL and Hyleak work on discrete random variables. Instead, our inference engine works on discrete and continuous (Gaussian) random variables, and produces the posterior distribution that can be used to compute mutual information and other quantitative information flow metrics [4, 12].

Stein and Staton proposed a Gaussian-based semantics to study exact conditioning through the lens of category theory [46]. They do not study the use of the semantics for privacy risks quantification on a subset of Python programs, or evaluate the efficiency of the semantics. The language that we present is based on the work [17], which introduces a probabilistic language with Gaussian mixture (denotational) semantics. Compared to that work, we investigate the applicability of Gaussian mixture semantics for privacy risk quantification of Python programs and introduce an operational semantics.

6 Conclusion

We have presented a novel inference engine for quantifying privacy risks in a subset of Python. The engine uses Gaussian mixture models to combine exact and approximate Bayesian inference. The engine supports continuous and discrete distributions to model attacker knowledge. We categorized the programs for which our engine produces exact results: programs involving Gaussian or discrete distributions, affine

Table 2 Comparison of leakage quantification tools. The column **Random variable type** indicates the type of random variable that the tool handles; either discrete (disc.), continuous (cont.) or both. **Analysis type** denotes whether the analysis returns exact results (Exact) or approximations (Approx.) computed by sampling/statistical mechanisms. **Analysis input** states the input to the tool, either a set of samples from the target system (set of samples), a custom language (custom), or a general purpose programming language such as Java, Scala or Python. Finally, **Analysis output** indicates what information the tool reports. It can be either leakage metrics—such as min-entropy (min-ent.), mutual information (mutual info.), Bayes risk, g-vulnerability metrics (g-vul), or the complete posterior attacker knowledge (Posterior); which can be used to compute the aforementioned metrics as well as to perform probability queries and distribution visualizations

Method	Random variable Type	Analysis Type	Analysis Input	Analysis Output
LeakiEst [5]	Disc./cont. (only output)	Approx.	set of samples	Mutual info., min-ent.
F-BLEAU [10]	Disc./cont.	Approx.	setof samples	Bayes risk
LEAVES [11]	Disc./cont.	Approx.	setof samples	g-vul. metrics
SPIRE [9]	Disc.	Exact	custom: PSI	Posterior
QUAIL [7]	Disc.	Exact	custom: QUAIL	Mutual Info.
HyLeak [8]	Disc.	Approx./Exact	custom: QUAIL 2.0	Mutual Info.
LeakWatch [6]	Disc./Cont.	Approx.	Java	Mutual Info., min-ent.
Privug (MCMC) [12]	Disc./Cont.	Approx.	Java/Scala/Python	Posterior
Privug (Gauss) [13]	Cont. (Gaussian)	Exact	Python (subset)	Posterior
Privug (this work)	Disc./cont.	Approx./Exact	Python (subset)	Posterior

transformations of Gaussian mixtures, and if-statements over discrete random variables. For programs involving non-Gaussian distributions or conditioning on inequalities over continuous random variables, we approximate distributions by matching their first and second moments to those of Gaussian distribution. We have presented an application of our engine to analyze privacy risks on public statistics as well as widely used privacy protection mechanisms: two differential privacy mechanisms (Gaussian and Laplace), randomized response, and attribute generalization. In the scalability evaluation, we have shown that our engine can analyze systems with thousands of random variables, which allows for analyzing systems involving sensitive data from thousands of individuals. This shows that our engine is applicable to realistic settings. All in all, this work provides a new point in understanding the trade-off among exact and approximate inference, program expressiveness and performance in quantification of privacy risks by means of Bayesian inference.

Future work includes investigating the impact of approximating non-Gaussian distributions as Gaussian mixtures on quantitative information flow metrics. Furthermore, we plan to explore inference engines based on other distributions from the exponential family; as they may allow for computing exact risks for a larger range of programs.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Article 29 Data Protection Working Party: Opinion 05/2014 on Anonymisation Techniques (2014). http://www.pdpjournals.com/docs/88197.pdf Accessed 2018-02-14
2. Elliot, M., Mackey, E., O'Hara, K., Tudor, C.: The Anonymisation Decision - Making Framework. University of Manchester, UKAN (2016)
3. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci. **9**(3–4), 211–407 (2014). https://doi.org/10.1561/0400000042
4. Alvim, M., Chatzikokolakis, K., McIver, A., Morgan, C., Palamidessi, C., Smith, G.: The Science of Quantitative Information Flow. Springer, Cham (2020). https://doi.org/10.1007/978-3-319-96131-6
5. Chothia, T., Kawamoto, Y., Novakovic, C.: A tool for estimating information leakage. In: CAV'13. LNCS, vol. 8044, pp. 690–695. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_47
6. Chothia, T., Kawamoto, Y., Novakovic, C.: Leakwatch: Estimating information leakage from Java programs. In: ESORICS'14. LNCS, vol. 8713. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_13
7. Biondi, F., Legay, A., Traonouez, L., Wasowski, A.: QUAIL: A quantitative security analyzer for imperative code. In: CAV'13, pp. 702–707. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_49
8. Biondi, F., Kawamoto, Y., Legay, A., Traonouez, L.: Hybrid statistical estimation of mutual information and its application to

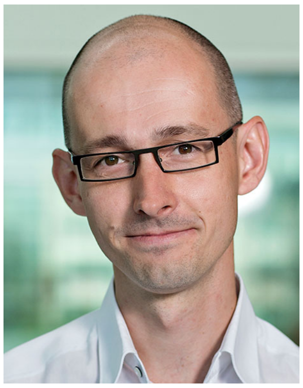
information flow. Formal Aspects Comput. **31**(2), 165–206 (2019). https://doi.org/10.1007/s00165-018-0469-z
9. Kucera, M., Tsankov, P., Gehr, T., Guarnieri, M., Vechev, M.T.: Synthesis of probabilistic privacy enforcement. In: CCS'17, pp. 391–408. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134079
10. Cherubin, G., Chatzikokolakis, K., Palamidessi, C.: F-BLEAU: fast black-box leakage estimation. In: SP'19, pp. 835–852. IEEE, New York, NY (2019). https://doi.org/10.1109/SP.2019.00073
11. Romanelli, M., Chatzikokolakis, K., Palamidessi, C., Piantanida, P.: Estimating g-leakage via machine learning. In: CCS'20. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3423363
12. Pardo, R., Rafnsson, W., Probst, C.W., Wasowski, A.: Privug: Using probabilistic programming for quantifying leakage in privacy risk analysis. In: ESORICS'21. LNCS, vol. 12973. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88428-4_21
13. Rønneberg, R.C., Pardo, R., Wasowski, A.: Exact and efficient bayesian inference for privacy risk quantification. In: Proceedings of the 21st International Conference Software Engineering and Formal Methods, SEFM'23. Lecture Notes in Computer Science, vol. 14323, pp. 263–281. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-47115-5_15
14. Pardo, R., Rafnsson, W., Steinhorn, G., Lavrov, D., Lumley, T., Probst, C.W., Ziedins, I., Wasowski, A.: Privacy with good taste - A case study in quantifying privacy risks in genetic scores. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2022 International Workshops, DPM 2022 and CBT 2022, Copenhagen, Denmark, September 26-30, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13619, pp. 103–119. Springer, ??? (2022). https://doi.org/10.1007/978-3-031-25734-6_7
15. Halvorsen, L., Steffensen, S.L., Rafnsson, W., Kulyk, O., Pardo, R.: How attacker knowledge affects privacy risks: An analysis using probabilistic programming. In: IWSPA@CODASPY 2022: Proceedings of the 2022 ACM on International Workshop on Security and Privacy Analytics, Baltimore, MD, USA, April 27, 2022, pp. 55–65. ACM, ??? (2022). https://doi.org/10.1145/3510548.3519380
16. Robert, C.P., George Casella: Monte Carlo Statistical Methods. Springer, New York, NY (2004). https://doi.org/10.1007/978-1-4757-4145-2
17. Randone, F., Bortolussi, L., Incerto, E., Tribastone, M.: Inference of probabilistic programs with moment-matching gaussian mixtures. Proc. ACM Program. Lang. 8(POPL), 1882–1912 (2024) https://doi.org/10.1145/3632905
18. Warner, S.L.: Randomized response: A survey technique for eliminating evasive answer bias. Journal of the American Statistical Association **60**(309), 63–69 (1965)
19. Sweeney, L.: Achieving k-anonymity privacy protection using generalization and suppression. Int. J. Uncertain. Fuzziness Knowl. Based Syst. 10(5), 571–588 (2002) https://doi.org/10.1142/S021848850200165X
20. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE'14, pp. 167–181. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2593882.2593900
21. Cover, T.M., Thomas, J.A.: Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing). Wiley-Interscience, USA (2006). https://doi.org/10.1002/047174882X
22. Eaton, M.L.: Multivariate Statistics: A Vector Space Approach. Lecture notes-monograph series. Institute of Mathematical Statistics, . (2007). https://doi.org/10.1177/002224378402100214
23. Koller, D., Friedman, N.: Probabilistic Graphical Models - Principles and Techniques. MIT Press, Cambridge MA (2009). https://doi.org/10.5555/1795555
24. Bishop, C.M.: Pattern Recognition and Machine Learning. Information science and statistics. Springer, New York (2006). https://doi.org/10.5555/1162264
25. Lo, J.: Finite-dimensional sensor orbits and optimal nonlinear filtering. IEEE Transactions on information theory **18**(5), 583–588 (1972). https://doi.org/10.1109/TIT.1972.1054885
26. Billingsley, P.: Convergence of Probability Measures. John Wiley & Sons, New York (2013). https://doi.org/10.1002/9780470316962
27. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014, pp. 167–181. ACM, ??? (2014). https://doi.org/10.1145/2593882.2593900
28. Sweeney, L.: k-anonymity: A model for protecting privacy. International Journal on Uncertainty, Fuzziness and Knowledge-based Systems **10**(5), 557–570 (2002). https://doi.org/10.1142/S0218488502001648
29. Kan, R., Robotti, C.: On moments of folded and truncated multivariate normal distributions. Journal of Computational and Graphical Statistics **26**(4), 930–934 (2017). https://doi.org/10.2139/ssrn.2748900
30. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf (2013)
31. Statistics Denmark. https://www.dst.dk/en. Accessed: 2023-06-23
32. Statistics New Zealand. https://www.stats.govt.nz/. Accessed: 2023-06-23
33. US Census Bureau. https://www.census.gov/. Accessed: 2023-06-23
34. Garfinkel, S.L., Abowd, J.M., Martindale, C.: Understanding database reconstruction attacks on public data. Commun. ACM **62**(3), 46–53 (2019). https://doi.org/10.1145/3287287
35. Dwork, C., Kohli, N., Mulligan, D.: Differential privacy in practice: Expose your epsilons! Journal of Privacy and Confidentiality **9**(2) (2019) https://doi.org/10.29012/jpc.689
36. Salvatier, J., Wiecki, T.V., Fonnesbeck, C.: Probabilistic programming in python using pymc3. PeerJ Computer Science 2, 55 (2016) https://doi.org/10.48550/arXiv.1507.08050
37. Rønneberg, R.C., Pardo, R., Wąsowski, A.: Exact and efficient bayesian inference for privacy risk quantification. In: Software Engineering and Formal Methods, pp. 263–281. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-47115-5_15
38. Gehr, T., Steffen, S., Vechev, M.: λPSI: exact inference for higher-order probabilistic programs. In: PLDI'20, pp. 883–897. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386006
39. Chothia, T., Guha, A.: A statistical test for information leaks using continuous mutual information. In: CSF'11, pp. 177–190. IEEE, New York, NY (2011). https://doi.org/10.1109/CSF.2011.19
40. Avi Pfeffer: Practical Probabilistic Programming. Manning Publications Co., New York, NY (2016). https://doi.org/10.5555/3033232
41. Chib, S., Greenberg, E.: Understanding the Metropolis-Hastings Algorithm, 10 https://doi.org/10.2307/2684568
42. Homan, M.D., Gelman, A.: The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. J. Mach. Learn. Res. **15**(1), 1593–1623 (2014). https://doi.org/10.5555/2627435.2638586
43. Gehr, T., Misailovic, S., Vechev, M.T.: PSI: exact symbolic inference for probabilistic programs. In: CAV'16. LNCS, vol. 9779, pp. 62–83. Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
44. Narayanan, P., Carette, J., Romano, W., Shan, C., Zinkov, R.: Probabilistic inference by program transformation in hakaru (system description). In: FLOPS'16. LNCS, vol. 9613, pp. 62–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29604-3_5

45. Saad, F.A., Rinard, M.C., Mansinghka, V.K.: SPPL: Probabilistic programming with fast exact symbolic inference. In: PLDI'21, pp. 804–819. ACM, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454078
46. Stein, D., Staton, S.: Compositional semantics for probabilistic programs with exact conditioning. In: LICS'21, pp. 1–13. IEEE, New York, NY (2021). https://doi.org/10.1109/LICS52264.2021.9470552

Rasmus Carl Rønneberg is a Doctoral Researcher in the research group TVA at Karlsruhe Institute of Technology. Before that he obtained his M.Sc. in Computer Science at the IT University of Copenhagen. His main research interests are formal methods, probabilistic programming and privacy protection.

Francesca Randone is a Postdoctoral Researcher in the Department of Mathematics, Informatics, and Geosciences at the University of Trieste, Italy. She earned her Ph.D. in Computer Science and Systems Engineering from the IMT School for Advanced Studies Lucca, Italy. Previously, she obtained her M.Sc. in Mathematics from the University of Trieste. Her research interests include inference methods and approximate semantics for probabilistic programming, with applications in systems modeling.

Raùl Pardo is an Associate Professor in the SQUARE group at the IT University of Copenhagen. His research is focused on developing rigorous techniques to design, analyze and build software to protect online privacy. His interests lie at the intersection of formal methods, online privacy and computer security. He has done research on privacy for social networks, Internet of Things (IoT) and data analytics. Within these topics, he is working on privacy risk analysis, formal verification of privacy legal requirements, probabilistic programming, and Bayesian data analysis. He holds a PhD degree from Chalmers University of Technology.



Andrzej Wąsowski is a Professor of Software Engineering at the IT University of Copenhagen, Denmark and the vice-president of the ETAPS association. He had previously worked at Aalborg University (Denmark) and as a visiting professor at RWTH Aachen (Germany), at INRIA Rennes (France), and at the University of (Waterloo), Ontario. His interests lie in software quality, reliability, and safety in high-stakes, high-value software projects. This encompasses semantic foundations and tool support for model-driven development, program analysis tools, testing methodologies, and processes for enhancing and maintaining software quality. Many of his projects involve collaborations with commercial or open-source partners, primarily in the domains of robotics and safety-critical embedded systems. Currently, he leads the Marie-Curie training network on Reliable AI for Marine Robotics (REMARO). Together with Thorsten Berger, he authored the Springer textbook on "Domain Specific Languages, Effective Modeling, Automation, and Reuse." He holds a PhD degree from the IT University of Copenhagen, Denmark (2005), and an MSc Eng degree from the Warsaw University of Technology, Poland (2000).