Probabilistic Bounds on Information Leakage

Ian Sweet  
Univ. of Maryland, College Park

José Manuel Calderón Trilla  
Galois, Inc.

Chad Scherrer  
Galois, Inc.

Michael Hicks  
Univ. of Maryland, College Park

Stephen Magill  
Galois, Inc.

Abstract—Quantitative information flow (QIF) is concerned with measuring the knowledge about secret data that is gained by observing the result of a computation over that data. QIF has important applications in the domain of privacy, as an increase in attacker knowledge corresponds to a decrease in the privacy of a user’s data. In this paper, we consider techniques for computing the Bayes Vulnerability of secret data due to answering a query. Our approach augments a baseline probabilistic abstract interpretation with both sampling and symbolic execution. The approach first computes a sound but imprecise upper bound on the vulnerability and then refines it. We prove this approach retains the soundness of the abstract interpretation. We also present detailed experimental results that quantify the precision and performance improvements of our techniques. We find augmenting imprecise abstract interpretation with our techniques can match the precision of precise abstract interpretation but with orders of magnitude better performance.

I. INTRODUCTION

As more data is created, collected, and analyzed, it is increasingly important to develop technology that enables productive use of this data while preserving privacy. One approach to this problem is to quantify the vulnerability of private data as computation over that data occurs. Doing so enables privacy policies that account for knowledge gained by a potential adversary when deciding whether further queries are safe to answer. One natural measure of vulnerability is the probability that an attacker can guess the value of the private data in a single try [39]. This metric is related to min entropy and has been used extensively in the quantitative information flow (QIF) literature. As knowledge is gained by observing the results of data analysis, the most likely values of private data become more apparent, so a privacy policy can define a threshold level of vulnerability that should not be exceeded [22].

A natural approach to estimating the vulnerability of private data following the execution of a query over that data is to model the query in a probabilistic programming language [16]. Such a language permits reasoning about the query as if it represented a joint distribution over random variables. These variables would represent the data we mean to keep private, where their distribution represents the adversary’s uncertainty about their true value. Unfortunately, as discussed in Section IX, most probabilistic languages are unsuitable for computing vulnerability because doing so would either be too expensive or potentially unsafe. By the latter we mean that the language could easily underestimate the vulnerability, leading to an unsafe security decision.

Work by Mardziel et al. [22] is a promising exception in being safe, at the least, and reasonably efficient in many cases. Their approach is to perform abstract interpretation [9] over the target query using an abstract domain called probabilistic polyhedron. Such a polyhedron (or a set of them [3]) can be used to characterize the joint distribution of the secrets prior to the query.1 To estimate the updated vulnerability, one analyzes the query using the prior distribution and revise it based on the output due to the actual secret value. Probabilistic polyhedra are designed to make it easy to extract the maximally probable point, i.e., the vulnerability, directly.

While Mardziel et al’s technique is useful, it still suffers from both imprecision and performance problems. In order to be sufficiently precise, it often requires the use of convex polyhedra [10] for numeric analysis. But then as the number of secret variables goes up the analysis becomes exponentially slower. Using non-relational intervals [8] instead can fix these performance problems, but doing so leads to an unacceptable loss of precision. This is because intervals can egregiously overapproximate the set of secret inputs that could lead to a particular query output.

In this paper we present a new approach that ensures a better balance of both precision and performance in vulnerability computation, augmenting Mardziel et al. with two new techniques. In both cases we begin by analyzing a query using the fast interval-based analysis, producing a representation of the posterior that estimates the adversary’s knowledge upon seeing the true output. Our first technique is then to use sampling to augment the result. In particular, we execute the query using points sampled from the posterior produced by the analysis. If the analysis were perfectly accurate, each sample’s execution result would match the true output. But since the analysis is overapproximate, some may not; these points are thus contributing to an overestimate of vulnerability. We therefore construct a Beta distribution from the outcomes of sampled points and use it to more accurately estimate the size of the support of the posterior, up to some level of

1Usefully, probabilistic polyhedra are not limited to representing only uniform distributions.
confidence. The more samples we take, the more accurate our estimate.

Our second technique is of a similar flavor, but uses symbolic reasoning to “magnify” the impact of a successful sample. In particular, we execute a query result-consistent sample concolically [37], thus maintaining a symbolic formula (called the path condition) that characterizes the set of variable valuations that would cause execution to follow the same path. We then count the number of possible solutions and use the count to boost the lower bound of the support in a sound manner. Both concolic execution and under approximation can be combined for further precision boosts.

The main results of this paper are as follows. First, we formally describe our two techniques (Sections V and VI) as extensions of Mardziel et al. (reviewed in Sections III and IV) and prove them sound. Second, we describe an implementation of these techniques in a domain-specific query language (Section VII). Finally, we present experimental results demonstrating the scalability and accuracy improvements obtained by using these techniques (Section VIII). Using our techniques, the implementation is able to quickly and accurately analyze queries that are intractable in other systems. In particular, our techniques provide similar precision to convex polyhedra while providing orders-of-magnitude better performance. As far as we are aware (Section IX), our approach constitutes the best balance of precision and performance proposed to date for estimating query vulnerability.

II. OVERVIEW

Suppose we have a query that takes as input some private data and produces a publicly-observable output. If we consider an attacker that has some initial belief about what the values of the private data might be, then observing the output allows the attacker to revise their belief to more tightly match the actual input values. Formally, we represent beliefs as probability distributions over the private data; we notate a distribution using symbol $\delta$, which is a function from secrets to probabilities. We use Bayesian inference as the revision operation. Thus the attacker in our model starts with some prior distribution capturing their belief about the private data. They then revise this belief based on the output, producing a posterior distribution.

What remains is to define an information flow metric that can characterize the loss of privacy due to a query output. We adopt Bayes Vulnerability: it characterizes the probability of the adversary guessing the secret in one try [39]. Formally, if $\delta$ is a distribution representing the adversary’s uncertainty about the secret, then its vulnerability is defined as follows

$$V(\delta) \triangleq \delta(\text{argmax}_x(\delta(x)))$$

That is, the vulnerability is probability of the most likely secret. From it we can compute min entropy: $-\log_2 V(\delta)$.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Range</th>
<th>Private?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShipID</td>
<td>Integer</td>
<td>1–10</td>
<td>No</td>
</tr>
<tr>
<td>NationID</td>
<td>Integer</td>
<td>1–20</td>
<td>No</td>
</tr>
<tr>
<td>Capacity</td>
<td>Integer</td>
<td>0–1000</td>
<td>Yes</td>
</tr>
<tr>
<td>Latitude</td>
<td>Integer</td>
<td>-900,000–900,000</td>
<td>Yes</td>
</tr>
<tr>
<td>Longitude</td>
<td>Integer</td>
<td>-1,800,000–1,800,000</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 1. The data model used in the evacuation scenario.

A. Motivating Scenario

To provide an overview of our approach, we will describe the application of our techniques to a scenario that involves a coalition of ships from various nations operating in a shared region. Suppose a natural disaster has impacted some islands in the region. Some number of individuals need to be evacuated from the islands due to medical needs, and it falls to a regional disaster response coordinator to determine how to accomplish this. While the coalition wants to collaborate to achieve these humanitarian aims, we assume that each nation also wants to protect their sensitive data—namely ship locations and capacity.

More formally, we assume the use of the data model presented in Figure 1, which tracks some number of ships, their coalition affiliation, the evacuation capacity of the ship, and its position, given in terms of latitude and longitude. We give latitude and longitude values as integer representations of decimal degrees fixed to four decimal places (so, 14.3579 decimal degrees would be encoded as the integer 143579).

To simplify syntax, we will refer to locations $L$, which are taken to be a pair of latitude and longitude. We write $L.y$ for latitude and $L.x$ for longitude. We will often index properties by ship ID, writing Capacity$(x)$ for the capacity associated with ship ID $x$, or Location$(x)$ for the location.

Our goal is to compute upper bounds on the information flowing to the disaster response coordinator. We assume that this coordinator initially has no knowledge of the positions or capabilities of the ships other than that they fall within certain expected ranges.

The evacuation problem is defined as follows:

Given a target location $L$, and number of people to evacuate $N$, compute a set of ship IDs $S$ such that the combined capacity $\sum_{s \in S} \text{Capacity}(s)$ is at least $N$.

Clearly if all members of the coalition share all of their data, then a solution is easy to compute. However, we want to support solutions that allow partner nations to maintain control over their data. One such solution is secure multi-party computation (MPC) [14], [42]. MPC allows for data owners to share their data with a third-party in a way that does not require the data owner to trust the third party. Another possible solution is to allow the coordinator to query about individual ships in a way that does not reveal precise values about their location or capacity, but rather suggests ranges of possibilities. Thus the coordinator could
Nearby$(s, L, d) = |\text{Location}(s).x - L.x| + |\text{Location}(s).y - L.y| \leq d$

$Nearby2(s, L_1, L_2, d) = Nearby(s, L_1, d) \lor Nearby(s, L_2, d)$

Figure 2. Queries used to implement resource allocation.

still complete the job but not learn too much about any individual ship. Figure 2 presents some such queries.

We can analyze the information flow properties of both approaches, but focus here on queries that provide bounds on location and capacity. Figure 3 gives an algorithm the response coordinator can follow to solve the evacuation problem. The procedure operates by maintaining upper and lower bounds on the capacity of each ship $i$ in the array $\text{berths}$. Each ship's bounds are updated based on the results of queries about its capacity and location. These queries aim to be privacy preserving, doing a sort of binary search to narrow in on the capacity of each ship in the operating area. The procedure completes once $\text{is\_solution}$ determines the minimum required capacity is reached.

B. Computing vulnerability with abstract interpretation

Using this procedure, what is revealed about the sensitive variables (location and capacity)? Consider a single $\text{Nearby}(s, L, d)$ query executed by this algorithm. At the start, the coordinator is assumed to know only that $s$ is somewhere within the operating region (per the ranges given in Figure 1), perhaps with all locations having equal probability. If the query returns true, the coordinator’s knowledge can be revised to know that $s$ must be within $d$ units of $L$ (using Manhattan distance). This situation is depicted in Figure 4. It shows a location $L$ and a diamond depicting the points at Manhattan distance $d$ from $L$. The true response rules out all points outside of the diamond as possible locations for $s$. To compute the vulnerability following the query, we compute the probability of the maximally probable point. Assuming the prior was uniformly distributed the posterior would be as probability of the maximally probable point. Assuming the vulnerability following the query, we compute the vulnerability of the posterior.

We are interested in static analysis algorithms for analyzing queries such as $\text{Nearby}(s, L, d)$, or entire procedures like the one in Figure 3, in order to estimate the vulnerability of secrets should queries be answered. Mardziel et al. [22] describe such an analysis. Their approach is to perform abstract interpretation over the target query using an abstract domain called a probabilistic polyhedron. This polyhedron (or a set of them) can be used to characterize the joint distribution of the secrets before and after a query result. This information can then be used to compute the vulnerability of the posterior.

Figure 4 gives an example probabilistic polyhedron using our $\text{Nearby}$ query to illustrate. An abstract element describes a set of probability distributions using a shape domain such as intervals [8], octagons [26], or polyhedra [10] paired with numeric bounds. The shape is used to approximate the set of points in the support of the distribution. The figure shows a diamond shape, which describes the positions for $x$ that would satisfy the query $\text{Nearby}(x, L_1, 4)$. This would be the approximation that a polyhedral or octagon shape domain would yield. The figure also shows the diamond overapproximated with a square, which is the approximation that the box or interval abstract domain would yield. A numeric quantity $s$ separately tracks lower and upper bounds on the number of points within the shape that are actually in the support (i.e., the points in the overapproximate shape that have non-zero probability). This improves precision and allows various operations that are important for sound use of the abstraction. To capture the probabilities involved, the abstraction also tracks bounds on the probability per point ($p$) and the total probability mass ($m$) contained in the region (where $m$ might be a more accurate estimate than $p \cdot s$ for the minimum or maximum bound). We will refer to the diamond or box portion of the abstraction as the shape and $s$, $p$, and $m$ as the ornaments.

Conceptually, a shape plus ornaments represents the set of distributions with support contained in the shape and probability per point, total mass, and number of support points satisfying $p$, $m$, and $s$ respectively. Soundness for this approach states that, after performing abstract interpretation, the actual posterior belief is contained in the set of distributions represented by the computed probabilistic polyhedron.

This probabilistic polyhedron-based analysis can be easily used to compute vulnerability. The main step required is normalization. The abstract shape paired with the ornaments captures the effect that learning the
query output has on attacker knowledge. Values that are impossible given the output have been removed from the distribution, and values that are less likely are scaled down. To compute bounds on the probability of private values, as the vulnerability computation requires, we must normalize the distribution by dividing by the probability mass. We are interested in sound upper bounds on probability, so we want to maximize $p/m$, where $p$ is the probability associated with a private data value and $m$ is the probability mass. This quotient is maximized when $p$ is maximized and $m$ is minimized, so we divide the upper bound on $p$ by the lower bound on $m$. Figure 5 includes examples of this maximum probability computation.

C. Improving Precision

In Figure 4, the parameters $s$, $p$, and $m$ are precise. However, as additional operations are performed, these quantities can accumulate imprecision. For example, suppose we combine this result with an abstraction corresponding to $\text{Nearby}(x, L_{2}, 4)$, obtaining an overapproximation of $\text{Nearby}(x, L_{1}, 4) \lor \text{Nearby}(x, L_{2}, 4)$. This corresponds to an

abstract join and requires updating the representation as shown in the top row of Figure 5. We first compute the join of the shapes (boxes in this case) and then take the minimum of the lower bounds and the maximum of the upper bounds for all the ornaments. The bottom row of Figure 5 shows on the left the exact result that can be obtained with a fully precise but costly representation. On the right of the bottom row, we show the most precise abstraction of this state in terms of probabilistic boxes. This is the best result we could expect to achieve with our abstract domain, but the techniques in [22] may fall short of this ideal. This can be seen in Figure 5, where the precise approach yields a maximum probability of 0.026, while the abstract representation yields max probability of 0.036.\(^2\)

By applying the precision recovery techniques described in this paper, we can work with abstract representations throughout analysis and then recover precision at any point. The effect of our techniques is shown on the right of Figure 5. Both focus on obtaining tighter lower bounds for $s$, the number of points in the support set. This allows us to update lower bounds on the probability mass $m$ since $m_{\text{min}}$ is at least $s_{\text{min}} \cdot p_{\text{min}}$ (each point has at least probability $p_{\text{min}}$ and there are at least $s_{\text{min}}$ of them).

The first technique we explore is under approximation, which consists of analyzing the query to determine points that must be in the support of the posterior distribution. We do this by using symbolic execution to explore some subset of the paths leading to the probabilistic polyhedron being analyzed. Under-approximation is sound in that whatever improved value it yields for $s_{\text{min}}$ is guaranteed to be a lower bound on $s$. Figure 5 demonstrates the impact this can have on the overall maximum probability (and thus vulnerability) computation. In the example, it improves the bound on maximum probability to 0.032 from 0.036.

The second technique we explore is sampling, depicted to the right of the arrow in Figure 5. Sampling chooses random points and evaluates the query on them to determine whether they are in the support of the posterior distribution for a particular query result. By tracking the ratio of points that produce the expected output, we can produce an estimate of $s$, whose confidence increases as we include more samples. We can also provide upper and lower bounds on $s$ that hold with a given confidence. This approach is depicted in the figure, where we conclude that $s \in [72, 81]$ with 90% probability after taking 1000 samples, improving our upper bound estimate to 0.028, which compares favorably with the precise estimate of 0.026.

These techniques are orthogonal and can also be used together to further increase precision. We show the effect of each technique in isolation as well as the combination in Section VIII which describes our experiments.

\(^2\)Since we are interested in upper bounds on maximum probability, lower values are better.
III. PRELIMINARIES: SYNTAX AND SEMANTICS

This section presents the core language—syntax and semantics—in which we formalize our approach to computing vulnerability. None of this section is new; it establishes definitions and concepts from earlier work. We begin by presenting the syntax of the language. Next, we present the semantics of the language in terms of how programs denote distributions of states. Finally, we present probabilistic polyhedra, which is a computable abstraction for the distribution-based semantics.

A. Core language

The programming language we use for queries is given in Figure 6. The language is essentially standard, apart from if statements. We limit the form of expressions to support our abstract interpretation-based semantics of probabilistic polyhedra, given below. In particular, arithmetic expressions are limited to linear forms; e.g., there is no division operator and multiplication of two variables is disallowed.

B. Semantics

We define the semantics of a program in terms of its effect on (discrete) distributions of states. States $\sigma$ are maps from variables to integers. Distributions $\delta$ are maps from states to positive real numbers, interpreted as probabilities (in range $[0, 1]$). The semantics is a denotational semantics that considers a program as a relation between distributions. In particular, semantics of statement $S$, written $[S]$, is a function of the form $\text{Dist} \rightarrow \text{Dist}$; we write $[S]\delta = \delta'$ to say that the semantics of $S$ maps input distribution $\delta$ to output distribution $\delta'$.

The semantics is given in Figure 7 along with definitions of relevant auxiliary operations. We write $[E]\sigma$ to denote the (integer) result of evaluating expression $E$ in $\sigma$.

3The notation $\sum_{x \in \mathcal{F} \phi \sigma}$ can be read $\rho$ is the sum over all $x$ such that formula $\phi$ is satisfied (where $x$ is bound in $\rho$ and $\phi$).

III. PRELIMINARIES: SYNTAX AND SEMANTICS

This section presents the core language—syntax and semantics—in which we formalize our approach to computing vulnerability. None of this section is new; it establishes definitions and concepts from earlier work. We begin by presenting the syntax of the language. Next, we present the semantics of the language in terms of how programs denote distributions of states. Finally, we present probabilistic polyhedra, which is a computable abstraction for the distribution-based semantics.

A. Core language

The programming language we use for queries is given in Figure 6. The language is essentially standard, apart from if statements. We limit the form of expressions to support our abstract interpretation-based semantics of probabilistic polyhedra, given below. In particular, arithmetic expressions are limited to linear forms; e.g., there is no division operator and multiplication of two variables is disallowed.

B. Semantics

We define the semantics of a program in terms of its effect on (discrete) distributions of states. States $\sigma$ are maps from variables to integers. Distributions $\delta$ are maps from states to positive real numbers, interpreted as probabilities (in range $[0, 1]$). The semantics is a denotational semantics that considers a program as a relation between distributions. In particular, semantics of statement $S$, written $[S]$, is a function of the form $\text{Dist} \rightarrow \text{Dist}$; we write $[S]\delta = \delta'$ to say that the semantics of $S$ maps input distribution $\delta$ to output distribution $\delta'$.

The semantics is given in Figure 7 along with definitions of relevant auxiliary operations. We write $[E]\sigma$ to denote the (integer) result of evaluating expression $E$ in $\sigma$.

3The notation $\sum_{x \in \mathcal{F} \phi \sigma}$ can be read $\rho$ is the sum over all $x$ such that formula $\phi$ is satisfied (where $x$ is bound in $\rho$ and $\phi$).
The and upper bounds on the probability mass are lower and upper bounds on the number of support points used in the probabilistic polyhedron. The quantities \( B \) set of linear inequalities with per-point probabilities varied somewhere in the range terminating, we will refer to \( P \rightarrow \mathcal{PD} \) as the concretization function for \( P \). We do not present the abstract semantics here; details can be found in Mardziel et al. \[22\] Importantly, this abstract semantics is sound:

**Theorem 1** (Soundness). For all \( S, P_1, P_2, \delta_1, \delta_2 \), if \( \delta_1 \in \gamma_P(P_1) \) and \( \langle S \rangle P_1 = P_2 \), then \( \langle S \rangle \delta_1 = \delta_2 \) with \( \delta_2 \in \gamma_P(P_2) \).

*Proof.* See Theorem 6 in Mardziel et. al \[22\].

Our implementation operates over finite sets (i.e., a disjunction) of probabilistic polyhedra, rather than a single polyhedron, applying the construction from the powerset domain \[3\]. This adds greater precision at somewhat higher performance cost. In what follows we present our approach as if using a single polyhedron \( P \) but give details about our implementation using finite sets of them in Section VII.

**IV. Computing Vulnerability: Basic procedure**

The key goal of this paper is to quantify the risk to secret information of running query over that information. This section explains the basic approach by which we can use probabilistic polyhedra to compute vulnerability, i.e., the probability of the most probable point of the posterior distribution. Improvements on this basic approach are given in the next two sections.

**A. Assumptions and notation**

Our convention will be to use \( C_1, s^\text{min}_1, s^\text{max}_1 \), etc. for the components associated with probabilistic polyhedron \( P_1 \). In the program \( S \) of interest, we assume that secret variables are in the set \( T \), so input states are written \( \sigma_T \), and we assume there is a single output variable \( r \). We assume that the adversary’s initial uncertainty about the possible values of the secrets \( T \) is captured by the probabilistic polyhedron \( P_0 \) (such that \( \forall \sigma \), \( \forall \delta \), \( \forall \delta' \), \( \forall \sigma' \)).

**B. Algorithm**

Computing vulnerability occurs according to the following procedure.

1) Perform abstract interpretation: \( \langle S \rangle P_0 = P \)

2) Given a secret input value \( \omega \), perform abstract conditioning to define \( P_{r=\omega} \equiv (P \land r=\omega) \).

The vulnerability \( z \) is the probability of the most likely state(s). When a probabilistic polyhedron represents one or more true distributions (i.e., the probabilities all sum to 1), the most probable state’s probability is bounded by \( P^\text{max} \). However, the abstract semantics does not always normalize the probabilistic polyhedron as it computes, so we need to scale \( P^\text{max} \) according to the total probability mass. To ensure that our estimate is on the safe side, we scale \( P^\text{max} \) using the minimum probability mass: \( z = \frac{P^\text{max}}{m^\text{min}} \).

\[4\]We often write \( P \land B \) and not \( P \mid B \) because \( P \) need not be normalized.
V. IMPROVING PRECISION WITH SAMPLING

We can improve the precision of the basic procedure using sampling. After computing \( P_T \) following the basic procedure we take the following additional steps:

1) Set counters \( \alpha \) and \( \beta \) to zero.
2) Do the following \( N \) times (for some \( N \), see below):
   a) "Run" the program by computing \( \mathbb{S}[\sigma_T = \delta] \). If there exists \( \sigma \in \text{support}(\delta) \) with \( \sigma(r) = o \) then increment \( \alpha \) by one. Otherwise increment \( \beta \) by one.
   b) "Run" the program by computing \( \mathbb{S}[\sigma_T = \delta] \). If there exists \( \sigma \in \text{support}(\delta) \) with \( \sigma(r) = o \) then increment \( \alpha \) by one. Otherwise increment \( \beta \) by one.
3) Compute a Beta distribution using \( \alpha \) and \( \beta \). With this distribution we can derive an interval \( [p_L, p_U] \) that bounds the likelihood that any given point in \( C_T \) is in the support of the true distribution. In general, a higher level of confidence \( \omega \) (where \( 0 \leq \omega \leq 1 \)) will require a higher \( N \) to produce a narrower interval.

Let result \( P_{T+} = P_T \) except that \( s_T^{\min} = p_L \cdot \#(C_T) \) and \( s_T^{\max} = p_U \cdot \#(C_T) \). We can then propagate these improvements to \( m^{\min} \) and \( m^{\max} \) by defining \( m_T^{\min} = p_T^{\min} \cdot s_T^{\min} \) and \( m_T^{\max} = p_T^{\max} \cdot s_T^{\max} \). Note that if \( m_T^{\min} > m_T^{\max} \) we just leave it alone, and do likewise if \( m_T^{\max} < m_T^{\min} \).

At this point we can compute the vulnerability as in the basic procedure, but using \( P_{T+} \) instead of \( P_T \).

There are several things to notice about this procedure. First, observe that in step 2b we “run” the program using the point distribution \( \sigma \) as an input; in the case that \( S \) is deterministic (has no \( \text{pif} \) statements) the output distribution will also be a point distribution. However, for programs with \( \text{pif} \) statements there are multiple possible outputs depending on which branch is taken by a \( \text{pif} \). We consider all of these outputs so that we can confidently determine whether the input state \( \sigma \) could ever cause \( S \) to produce result \( o \). If so, then \( \sigma \) should be considered part of \( P_{T+} \); if not, then we can safely rule it out (i.e., it’s part of the overapproximation).

Second, we only update the size parameters of \( P_{T+} \); we make no changes to \( p_T^{\min} \) and \( p_T^{\max} \). This is because our sampling procedure only determines whether it’s possible for an input state to produce the expected output. The probability that an input state produces an output state is already captured (soundly) by \( P_T \) so we do not change that. This is useful because the approximation of \( P_T \) does not degrade with the use of the interval domain in the way the approximation of the size degrades (as illustrated in Figure 5). Using sampling is an attempt to regain the precision lost on the size component (only).

Finally, the confidence we have that sampling has accurately assessed which input states are in the support is orthogonal to the probability of any given state. In particular, \( P_T \) is an abstraction of a distribution \( \delta_T \), which is a mathematical object. Confidence \( \omega \) is a measure of how likely it is that our abstraction (or, at least, the convex polyhedral part of it) is accurate.

We can formally state that our sampling procedure is sound:

**Theorem 2** (Sampling is Sound).
If \( \delta_0 \in \gamma_E(P_0) \), \( \mathbb{S}[P_0 = P] \), and \( [S]\delta_0 = \delta \) then
\[
\delta_T \in \gamma_E(P_{T+}) \text{ with confidence } \omega
\]
where
\[
\delta_T \overset{\text{def}}{=} \delta \land (r = o) \mid T
\]
\[
P_T \overset{\text{def}}{=} P \land (r = o) \mid T
\]
\[
P_{T+} \overset{\text{def}}{=} P_T \text{ sampling revised with confidence } \omega.
\]

The proof is in the appendix.

VI. IMPROVING PRECISION WITH CONCOLIC EXECUTION

Another approach to improving the precision of a probabilistic polyhedron \( P \) is to use concolic execution. The idea here is to “magnify” the impact of a single sample to soundly increase \( s^{\min} \) by considering its execution symbolically. More precisely, we concretely execute a program using a particular secret value, but maintain symbolic constraints about how that value is used. This is referred to as concolic execution [37]. We use the collected constraints to identify all points that would induce the same execution path, which we can include as part of \( s^{\min} \).

We begin by defining the semantics of concolic execution, and then show how it can be used to increase \( s^{\min} \) soundly.

A. (Probabilistic) Concolic Execution

Concolic execution is expressed as rewrite rules defining a judgment \( \langle \Pi, S \rangle \rightarrow^{\varpi} \langle \Pi', S' \rangle \). Here, \( \Pi \) is pair consisting of a concrete state \( \sigma \) and symbolic state \( \gamma \). The latter maps variables \( x \in \text{Var} \) to symbolic expressions \( E \) which extend expressions \( E \) with symbolic variables \( \alpha \). This judgment indicates that under input state \( \Pi \) the statement \( S \) reduces to statement \( S' \) and output state \( \Pi' \) with probability \( p \), with path condition \( \pi \). The path condition is a conjunction of boolean symbolic expressions \( B \) (which are just boolean expressions \( B \) but altered to use symbolic expressions \( E \) instead of expressions \( E \)) that record which branch is taken during execution. For brevity, we omit \( \pi \) in a rule when it is true.

The rules for the concolic semantics are given in Figure 8. Most of these are standard, and deterministic (the probability annotation \( p \) is 1). Path conditions are recorded for if and while, depending on the branch taken. The semantics of \( \text{pif} \ q \) then \( S_1 \) else \( S_2 \) is non-deterministic: the result is that of \( S_1 \) with probability \( q \), and \( S_2 \) with probability \( 1 - q \). We write \( \gamma(B) \) to substitute free variables \( x \in B \) with their mapped-to values \( \gamma(x) \) and then simplify the result as much as possible. For example, if \( \gamma(x) = \alpha \) and \( \gamma(y) = 2 \), then \( \gamma(x + y + 3) = \alpha > 5 \). The same goes for \( \gamma(E) \).
We define a complete run of the concolic semantics with the judgment $(Π, S) \Downarrow^p_\Pi'$, which has two rules:

$$(Π, skip) \Downarrow^1_\Pi$$

$$(Π, S) \Downarrow^p_Π (Π', S') \Downarrow^q_Π (Π', S'') \Downarrow^r_Π$$

A complete run’s probability is thus the product of the probability of each individual step taken. The run’s path condition is the conjunction of the conditions of each step.

The path condition $π$ for a complete run is a conjunction of the (symbolic) boolean guards evaluated during an execution. $π$ can be converted to disjunctive normal form (DNF), and given the restrictions of the language the result is essentially a set of convex polyhedra over symbolic variables $α$.

B. Improving precision

Using concolic execution, we can improve our estimate of the size of a probabilistic polyhedron as follows:

1) Randomly select an input state $σ_T \in γ_C(C_T)$ (recall that $C_T$ is the polyhedron describing the possible valuations of secrets $T$).

2) Set $Π = (σ_T, γ_T)$ where $γ_T$ maps each variable $x \in T$ to a fresh symbolic variable $α_x$. Perform a complete concolic run $(Π, S) \Downarrow^p_\Pi (σ', γ')$. Make sure that $σ'(r) = o$, i.e., the expected output. If not, select a new $σ_T$ and retry. Give up after some number of failures $N$.

3) After a successful concolic run, convert path condition $π$ to DNF, where each conjunctive clause is a polyhedron $C_i$. Also convert uses of disequality ($≤$ and $≥$) to be strict (< and >).

4) Let $C = C_0 \cap (\bigcup_i C_i)$; that is, it is the join of each of the polyhedra in DNF($π$) “intersected” with the original constraints. This captures all of the points that could possibly lead to the observed outcome along the concolically executed path. Compute $n = #(C)$.

Let $P_T = P_T$ except define $s_{\min}^T = n$ if $s_j^0 < n$ and $m_{\min}^T = p_{\min}^T = n$ if $m_j^0 < p_{\min}^T$. (Leave them as is, otherwise.)

Theorem 3 (Concolic Execution is Sound).
If $δ_0 ∈ γ_0(P_0)$, $⟨S⟩ P_0 = P$, and $[S]δ_0 = δ$ then

$δ_T ∈ γ_0(P_{T+})$

where

$δ_T \overset{\text{def}}{=} δ ∧ (r = o) | T$

$P_T \overset{\text{def}}{=} P ∧ (r = o) | T$

$P_{T+} \overset{\text{def}}{=} P_T$ concolically revised.

The proof is in the appendix.

VII. Implementation

We have implemented our approach as an extension of Mardziel et al. [22], which is written in OCaml. The analysis represents numeric domains $C$ using an OCaml interface to the Parma Polyhedra Library. The procedure $\#(C)$ is implemented by LattE. Support for arbitrary precision and exact arithmetic (e.g., for manipulating $m_{\min}^T$, $p_{\min}^T$, etc.) is provided by the migmp OCaml interface to the GMP library.

Our implementation deviates from the formal presentation in two ways. First, we have only implemented the analyses for the interval numeric domain [8]. As a matter of practicality, intervals are the least precise domain and therefore leave the most room for dynamic analyses to improve precision. The theory fully applies to any numeric abstract domain. Second, rather than maintaining a single probabilistic polyhedron $P$, we maintain a powerset of polyhedra [3], i.e., a finite disjunction. Doing so results in a more precise handling of join points in the control flow. It is straightforward to lift the various operations we have described to the powerset domain.

We now elaborate on how we implement sampling, and how we combine sampling and concolic execution together, in our implementation, to improve precision yet further.

A. Sampling

The use of sampling to infer properties of a distribution is common in Bayesian inference. In order for the sampling to be statistically sound it is important that our random samples are sampled uniformly from the polyhedron. We use Gibbs sampling to achieve this in our implementation. In the context of $N$ dimensional polyhedra, Gibbs sampling for $k$ samples can be described as follows:

1) Pick a random point within the polyhedron: $p_0 = (x_0^0, x_0^1, ..., x_0^n)$

2) Starting with $n = 1$, set $p_k = (x_0^k, x_1^k, ..., x_n^k, ...)$ where $x_i^n$ is a random value from within the bounds of dimension $n$

3) Determine whether $p_k$ falls within the support

4) Repeat for the number of desired samples, cycling through the number of dimensions

Gibbs sampling is a way to reducing taking one sample in $N$ dimensions to $N$ samples in one dimension. Because the samples are correlated with each other there is a 'burn-in' period where the choice of initial value has a disproportionate effect on the results. However, because of
the guarantee of uniformity it does not affect the statistical significance when taking many samples [18].

The above describes how to sample from a single polyhedra. When sampling from the powerset abstraction we must take into account the relative sizes of the constituent polyhedra when sampling. In order to maintain uniform sampling, each constituent polyhedra is samples in proportion to its contribution to the overall size of the powerset. For example, a powerset contains three polyhedra A, B, and C, with sizes 50, 30, and 20, respectively. In order to produce N samples of the powerset, we sample \( \frac{50}{100} N \) times from A, \( \frac{30}{100} N \) from B, and \( \frac{20}{100} N \) from C. This ensures uniformity over the entire space represented by the powerset.

**B. Combining Sampling with Concolic Execution**

Sampling can be used in addition to concolic execution, rather than just choosing one or the other. The presence of a sound under-approximation means that it is unnecessary (and counter-productive) to sample from the under-approximating region. When taking samples we first check whether it falls within the under-approximating region, if so we discard that sample and continue sampling.

The resulting samples are of the spaces around the under-approximating region \( \alpha \), and therefore the size of \( \alpha \) must be subtracted from the size of the overall region when performing the calculations for the beta-distribution. We then add back the size of \( \alpha \) when calculating the bounds on the total number of points in the support, as all the points within \( \alpha \) are in the support by the definition of the under-approximation.

The bulk of the implementation is written in 75 lines of OCaml code with and the addition of a `sample_region` function in the API.

Our code implements Gibbs sampling and the propagation of the improvements to other parts of the probabilistic polyhedra. We delegate the calculation of the beta distribution and its corresponding credible interval to the pareto OCaml library, which in turns uses the GNU Scientific Library.

**VIII. EXPERIMENTS**

To evaluate the benefits of our techniques, we applied them to queries based on the evacuation problem outlined in Section II. We found that while the baseline technique of Mardziel et al. [22] can yield precise answers when computing vulnerability, our new techniques can achieve close to the same level of precision far more efficiently.

**A. Experimental Setup**

For our experiments we analyzed queries similar to `Nearby(s, L, d)` from Figure 2. We generalize the `Nearby` query to accept a set of locations `L`—the query returns true if `s` is within `d` units of any one of the islands having location \( l \in L \). In our experiments we fix `d = 100`. We also analyze the execution of the resource allocation algorithm of Figure 3 directly; we discuss this in Section VIII-C.

We measure the time it takes to compute the vulnerability (i.e., the probability of the most probable point) following each query. In our experiments, we consider a single ship `s` and set its coordinates so that it is always in range of some island in `L`, so that the concrete query result returns `true`. We measure the vulnerability following this query result starting from a prior belief that the coordinates of `s` are uniformly distributed with \( 0 \leq \text{Location}(s).x \leq 1000 \) and \( 0 \leq \text{Location}(s).y \leq 1000 \).

In our experiments, we varied several experimental parameters: **analysis method** (either P, I, CE, S, or CE+S); **query complexity** `c`; **AI precision level** `p`; and **number of samples** `n`. We describe each in turn.

**Analysis method**: We compared five different techniques for computing vulnerability:

- **P**: Abstract interpretation (AI) with convex polyhedra for shape domain `C` (Section IV),
- **I**: AI with intervals for `C` (Section IV),
- **CE**: AI with intervals augmented with concolic execution (Section VI),
- **S**: AI with intervals augmented with sampling (Section V), and
- **CE+S**: AI with intervals augmented with both sampling and concolic execution (Section VII-B)

The first two techniques are due to Mardziel et al. [22], where the former uses convex polyhedra and the latter uses intervals (aka boxes) for the underlying polygons. In our experiments we tend to focus on P since I’s precision is unacceptably poor (e.g., often vulnerability = 1).

**Query complexity**: We consider queries with different `L`; we say we are increasing the complexity of the query as `L` grows larger. Let `c = |L|`; we consider `1 \leq c \leq 5`, where larger `L` include the same locations as smaller ones. We set each location to be at least `2 \cdot d` Manhattan distance units away from any other island (so diamonds like those in Figure 4 never overlap).

**Precision**: The precision parameter `p` bounds the size of the powerset abstract domain at all points during abstract interpretation. This has the effect of forcing joins when the powerset grows larger than the specified precision. As `p` grows larger, the results of abstract interpretation are likely to become more precise (i.e., vulnerability gets closer to the true value). We considered `p` values of 1, 2, 4, 8, 16, 32, and 64.

**Samples taken**: For the latter three analysis methods, we varied the number of samples taken `n`. For analysis CE, `n` is interpreted as the number of samples to try per polyhedron before giving up trying to find a “valid sample.”\(^5\) For analysis S, `n` is the number of samples, distributed proportionally across all the polyhedra in the powerset. For analysis CE+S, `n` is the combination of the two. We considered sample size values of 1,000 — 50,000 in increments of 1,000. We always compute an interval with

\(^5\)This is the `N` parameter from section VI.
\( \omega = 99.9\% \) confidence (which will be wider when fewer samples are used).

**System description:** We ran experiments varying all possible parameters. For each run, we measured the total execution time (wall clock) in seconds to analyze the query and compute vulnerability. All experiments were carried out on a MacBook Air with OSX version 10.11.6, a 1.7GHz Intel Core i7, and 8GB of RAM.

**B. Results**

Figures 9–11 measure vulnerability (y-axis) as a function of time (x-axis) for each analysis.\(^6\) These three figures characterize three interesting “zones” in the space of complexity and precision. The results for method I are not shown in any of the figures. This is because I always produces a vulnerability of 1. The refinement methods (CE, S, and CE+S) are all over the interval domain, and should be considered as “improving” the vulnerability of I.

In Figure 9 we fix \( c = 1 \) and \( p = 1 \). In this configuration, baseline analysis P can compute the true vulnerability in \( \sim 0.95 \) seconds. Analysis CE is also able to compute the true vulnerability, but in \( \sim 0.19 \) seconds. Analysis S is able to compute a vulnerability to within \( \sim 5 \cdot e^{-6} \) of optimal in \( \sim 0.15 \) seconds. These data points support two key observations. First, even a very modest number of samples improves vulnerability significantly over just analyzing with intervals. Second, concolic execution is only slightly slower and can achieve the optimal vulnerability. Of course, concolic execution is not a panacea. As we will see, a feature of this configuration is that no joins take place during abstract interpretation. This is critical to the precision of the concolic execution.

In Figure 10 we fix \( c = 2 \) and \( p = 4 \). In contrast the the configuration of Figure 9, the values for \( c \) and \( p \) in this configuration are not sufficient to prevent all joins during abstract interpretation. This has the effect of taking polygons that represent individual paths through the program and joining them into a single polygon representing many paths. We can see that this is the case because baseline analysis P is now achieving a better vulnerability than CE. However, one pattern from the previous configuration persists: all three refinement methods (CE, S, and CE+S) are all over the interval domain, and should be considered as “improving” the vulnerability of I.

In Figure 11 we fix \( c = 5 \) and \( p = 32 \). This configuration magnifies the effects we saw in Figure 10. Similarly, in this configuration there are joins happening, but the query is much more complex and the analysis is much more precise. In this figure, we label the X axis as a log scale over time. This is because analysis P took over two minutes to complete, in contrast the longest-running refinement method, which took less than 6 seconds. The relationship between the refinement analyses is similar to the previous configuration. The key observation here is that, again, all three refinement analyses achieve within \( \sim 3 \cdot e^{-5} \) of P,

\(^6\)These are best viewed on a color display.
but this time in 4% of the time (as opposed to $\frac{1}{4}$ in the previous configuration).

Figure 12 makes more explicit the relationship between refinements (CE, S, CE+S) and P. We fix $n = 50,000$ (the maximum here), and $p = 64$ (the maximum). We can see that as query complexity goes up, P gets exponentially slower, while CE, S, and CE+S slow at a much lower rate, while retaining (per the previous graphs) similar precision.

C. Evacuation Problem

We conclude this section by briefly discussing an analysis of an execution of the resource allocation algorithm of Figure 3. In our experiment, we set the number of ships to be three, where two were in range $d = 300$ of the evacuation site, and their sum-total berths (500) were sufficient to satisfy demand at the site (also 500). For our analysis refinements we set $n = 1000$. Running the algorithm, a total of seven pairs of Nearby and Capacity queries were issued (Figure 2). In the end, the algorithm selects two ships to handle the evacuation.

Table I shows the time to execute the algorithm using the different analysis methods, along with the computed vulnerability—this latter number represents the coordinator’s view of the most likely nine-tuple of the private data of the three ships involved (x coordinate, y coordinate, and capacity for each). We can see that, as expected, our refinement analyses are far more efficient than baseline P, and far more precise than baseline I. The CE methods are precise but slower than S. This is because of the need to count the number of points in the DNF of the concolic path conditions, which is expensive.

IX. Related Work

A. Quantifying Information Flow

Formal approaches to security based on noninterference [13] are often too strong: most programs release some information. A rich literature has emerged on characterizing application-specific situations when it is safe to violate noninterference; such characterizations are variously called declassification or information release policies [35]. These policies implicitly take it that some secret information can be released. One strand of research aims to quantify information that a program may release, or has released, and then use that quantification as a basis for policy.

One question is what measure of information release should be used. Past work largely considers information theoretic measures, including Bayes vulnerability [40] and Bayes risk [5], Shannon entropy [38], and guessing entropy [23]. The $g$-vulnerability framework [1] was recently introduced to express measures having richer operational interpretations, and subsumes other measures.

Our work focuses on Bayes Vulnerability, which is related to min entropy. Vulnerability is appealing operationally: As Smith [40] explains, it estimates the risk of the secret being guessed in one try. However, it is more challenging to compute, especially when the prior is non-uniform, because it requires determining the most probable secret value. Other work that analyzes programs to quantify the information they release has focused on other, easier-to-compute metrics, such as Shannon entropy and channel capacity, and assumes that priors conform to uniform distributions [2], [20], [21], [24], [29]. Like Mardziel et al. [22], we quantify dynamic information release, i.e., due to a particular execution’s output, rather than a static estimate, i.e., as an expectation over all possible outputs. Köpf and Basin [19] originally proposed this idea, and Mardziel et al. were the first to implement it (to audit a sequence of queries).

B. Probabilistic Programming Languages

Our work is fundamentally related to work on probabilistic programming languages [16], which are often used to express machine learning tasks. A probabilistic program is essentially a lifting of a normal program operating on single values to a program operating on distributions of values. As a result, the program represents a joint distribution. Probabilistic languages are distinguished by what operations they support over this joint distribution, how they perform them, and how efficient and precise the results are.

The primary goal of our work is to determine the probability of the most likely value in the posterior probability distribution that results from running a sensitive
query. Depending on the query, this corresponds to either the maximum likelihood estimation (MLE) problem or the maximum a-posteriori probability (MAP) problem in probabilistic programming. Not all implementations support computation of MLE and MAP, but among those that do there are generally two approaches: exact inference and approximate methods.

Approximate inference systems, such as Infer.net [27] do not provide sound upper bounds on posterior probability and can get stuck in local optima when the posterior is not convex. This is in contrast to our approach which has the ability to produce sound upper bounds. Furthermore, even when sampling is used to produce approximate bounds, the approach we describe can provide a confidence measure to quantify the amount of certainty regarding the result.

Other probabilistic language implementations based on partial sampling [15], [31] or full enumeration [34] of the state space are unsuitable in our setting. Such tools are either too inefficient or too imprecise. Works based on smarter representations of probability distributions are promising alternatives. PSI [12] supports exact inference via computation of precise symbolic representations of posterior distributions. However, that system does not implement algorithms for computing MAP or MLE over those representations. Projects based on algebraic decision diagrams [6], graphical models [25], and factor graphs [4], [33] translate programs into convenient structures and take advantage of efficient algorithms for their manipulation or inference, in some cases supporting MAP or MLE queries (e.g. [30], [32]).

Our implementation for probabilistic computation and inference differs from these existing works in two main ways. Firstly, we are capable of approximation and hence can trade off precision for performance, while maintaining soundness in terms of a strong security policy. The second difference is the nature of our representation of probability distributions. Our work is based on numerical abstractions: intervals [8], octagons [26], and polyhedra [10]. These abstractions are especially well suited for analysis of imperative programs with numeric variables and linear conditionals. Other probabilistic languages might serve as better choices when nominal, rather than numeric, variables are used in queries. The comparative study of the power and effectiveness of various representations in probabilistic computation is a topic of our ongoing research.

C. Abstract Interpretation

Building on Mardziel et al [22], our work uses abstract interpretation to overapproximate the distribution represented by the program. The overapproximation is sound; that is, it always includes the true distribution for all supported program operations (including conditioning, which is important for quantifying the information revealed due to a query result). The overapproximation may also describe other distributions, but not in a way that would cause it to underestimate the true min entropy. In particular, to compute vulnerability, we need to compute the maximum probability of any point in the support, which is easy to do with our representation. By considering all distributions when we do so we end up computing an upper bound on the actual vulnerability, which is the safe outcome from a security perspective.

A few other works have also focused on abstract interpretation, or related techniques, for reasoning about probabilistic programs. Monniaux [28] defines an abstract domain for distributions. Smith [41] describes probabilistic abstract interpretation for verification of quantitative program properties. Cousot [11] unifies these and other probabilistic program analysis tools. However, these do not deal with sound distribution conditioning, which is crucial for belief-based information flow analysis. Work by Sankaranarayanan et al [36] uses a combination of techniques from program analysis to reason about distributions (including abstract interpretation), but the representation does not support efficient retrieval of the maximal probability, needed to compute vulnerability.

D. Symbolic Execution and Sampling

Köpf and Rybalchenko also use symbolic execution in order to learn about the lower bounds of leakage, in a similar manner to the way that we use concolic execution [20]. Their use of randomized sampling is to control the number of preimages that they must approximate (since the number of total preimages can be prohibitively large). However, their work does not use randomized sampling to improve the upper bounds in the way we presented above.

Our use of sampling to improve the bounds on our analysis depends on statistical inference. The current implementation uses Bayesian inference via a credible interval over a beta distribution. One alternative would be to use the equivalent frequentist inference by calculating a more traditional confidence interval. For our application the differences are mostly philosophical. Jaynes and Kempthorne provide background on the differences between the two approaches and describe when Bayesian inference might be preferred [17].

X. Conclusions

Quantitative information flow is concerned with measuring the knowledge about secret data that is gained by observing the answer to a query. This paper has presented a combination of static analysis—using probabilistic abstract interpretation—and underapproximation and concolic execution to compute high-confidence upper bounds on information flow more precisely and efficiently than past work. Experimental results show dramatic improvements in overall precision and/or performance compared to abstract interpretation alone. As next steps, we plan to integrate static analysis and sampling more closely so as to avoid precision loss at decision points in programs. We also look to extend programs to be able to store random choices in variables, to thereby implement more advanced probabilistic structures.
REFERENCES


All links were last followed on October 5, 2014.

APPENDIX

Here we restate the soundness theorems for our techniques, and include their proofs.
Theorem 2 (Sampling is Sound).
If $\delta_0 \in \gamma_P(P_0)$, $\langle S \rangle P_0 = P$, and $[S]\delta_0 = \delta$ then
\[ \delta_T \in \gamma_P(P_{T+}) \text{ with confidence } \omega \]
where
\[ \delta_T \overset{\text{def}}{=} \delta \land (r = o) \mid T \]
\[ P_T \overset{\text{def}}{=} P \land (r = o) \mid T \]
\[ P_{T+} \overset{\text{def}}{=} P_T \text{ sampling revised with confidence } \omega. \]

Proof. Suppose we have some $\delta_0 \in \gamma_P(P_0)$ whereby $[S]\delta_0 = \delta$. We want to prove that $\delta_T \in \gamma_P(P_{T+})$. By Definition 2, this means we must show that
(1) $\text{support}(\delta_T) \subseteq \gamma_T(C_{T+})$
(2) $s_{T+}^{\text{min}} \leq |\text{support}(\delta_T)| \leq s_{T+}^{\text{max}}$
(3) $m_{T+}^{\text{min}} \leq \|\delta_T\| \leq m_{T+}^{\text{max}}$
(4) $\forall \sigma \in \text{support}(\delta_T). p_{T+}^{\text{min}} \leq \delta_T(\sigma) \leq p_{T+}^{\text{max}}$

Our proof goes as follows. First, we know that $\delta_T \in \gamma_T(C_T)$ by Theorem 1, Lemma 15 and Lemma 7 of Mardziel et al. By Definition 2, this means
(a) $\text{support}(\delta_T) \subseteq \gamma_T(C_T)$
(b) $s_{T+}^{\text{min}} \leq |\text{support}(\delta_T)| \leq s_{T+}^{\text{max}}$
(c) $m_{T+}^{\text{min}} \leq \|\delta_T\| \leq m_{T+}^{\text{max}}$
(d) $\forall \sigma \in \text{support}(\delta_T). p_{T+}^{\text{min}} \leq \delta_T(\sigma) \leq p_{T+}^{\text{max}}$

So (1) and (4) follow directly from (a) and (d), since $p_{T+}^{\text{min}} = m_{T+}^{\text{min}}$, $p_{T+}^{\text{max}} = m_{T+}^{\text{max}}$, and $C_T = C_{T+}$. To prove (2), we argue as follows. Let $p = \frac{|\text{support}(\delta_T)|}{\#(C_T)}$, which represents the probability that a randomly selected point from $C_T$ is in support(\delta_T). From the computed credible interval over the Beta distribution, we have that
\[ p \in [p_L, p_U] \text{ with confidence } \omega. \]
So such
\[ p_L \leq p \leq p_U \]
\[ p_L \cdot \#(C_T) \leq |\text{support}(\delta_T)| \leq p_U \cdot \#(C_T) \]
\[ s_{T+}^{\text{min}} \leq |\text{support}(\delta_T)| \leq s_{T+}^{\text{max}} \]
\[ m_{T+}^{\text{min}} \leq \|\delta_T\| \leq m_{T+}^{\text{max}} \]
\[ p_{T+}^{\text{min}} \leq \delta_T(\sigma) \leq p_{T+}^{\text{max}} \]
which is the desired result.

To prove (3), first consider that if $m_{T+}^{\text{min}} = m_{T+}^{\text{min}}$ then the first half of (3) follows from the first half of (c). Otherwise, we have that $m_{T+}^{\text{min}} = p_{T+}^{\text{min}} \cdot s_{T+}^{\text{min}}$. Then we can reason the first half of (3) holds using the following reasoning:
\[ m_{T+}^{\text{min}} \leq p_{T+}^{\text{min}} \cdot \text{support}(\delta_T) \]
\[ m_{T+}^{\text{min}} \leq p_{T+}^{\text{min}} \cdot |\text{support}(\delta_T)| \]
\[ m_{T+}^{\text{min}} \leq \|\delta_T\| \]
\[ m_{T+}^{\text{min}} \leq \|\delta_T\| \]
\[ m_{T+}^{\text{min}} \leq \|\delta_T\| \]
\[ m_{T+}^{\text{min}} \leq \|\delta_T\| \]

We can prove the soundness of $m_{T+}^{\text{max}}$ (the other half of (3)) with similar reasoning. $\square$

Theorem 3 (Concolic Execution is Sound).
If $\delta_0 \in \gamma_P(P_0)$, $\langle S \rangle P_0 = P$, and $[S]\delta_0 = \delta$ then
\[ \delta_T \in \gamma_P(P_{T+}) \]